Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Avaliando Suites Automaticamente Geradas para Validação de Refatoramentos

Indy Paula Soares Cordeiro e Silva

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Metodologia e Técnicas da Computação

Everton Leandro Galdino Alves
Patrícia Duarte de Lima Machado
(Orientadores)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

S586a Silva, Indy Paula Soares Cordeiro e.

Avaliando suítes automaticamente geradas para validação de refatoramentos / Indy Paula Soares Cordeiro e Silva. ï Campina Grande, 2018.

91 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2018.

"Orientação: Everton Leandro Galdino Alves, Patrícia Duarte de Lima Machado".

Referências.

1. Refatoramento. 2. Teste de Regressão. 3. Ferramentas de Geração de Testes. I. Alves, Everton Leandro Galdino. II. Machado, Patrícia Duarte de Lima. III. Título.

CDU 004.415.53(043)

Resumo

Refatoramentos normalmente exigem testes de regressão para verificar se as mudanças aplicadas ao código, preservaram o comportamento original do mesmo. Geralmente, é difícil definir um conjunto de testes que seja efetivo para esta tarefa, uma vez que o refatoramento não é frequentemente aplicado em etapas isoladas. Além disso, as edições de refatoramento podem ser combinadas com outras edições no código. Nesse sentido, a geração de casos de teste pode contribuir para essa tarefa, analisando sistematicamente o código e fornecendo uma ampla gama de casos de teste que abordam diferentes construções. No entanto, uma série de estudos apresentados na literatura mostram que as ferramentas atuais ainda não são eficazes com relação à detecção de faltas, particularmente faltas de refatoramento. Com base nisto, apresentamos dois estudos empíricos que aplicaram as ferramentas Randoop e Evosuite para gerar suites de testes de regressão, com foco na edição de refatoramento do tipo *extract method*. Com base nos resultados dos estudos, identificamos fatores que podem influenciar o desempenho das ferramentas para efetivamente testar a edição. Para validar nossos achados, apresentamos um conjunto de modelos de regressão que associam a presença desses fatores à capacidade do conjunto de testes, de detectar faltas relacionadas à edição de refatoramento. E por fim, apresentamos a REFANALYZER, que é uma ferramenta que objetiva ajudar os desenvolvedores a decidir quando confiar em suites geradas automaticamente, para validar refatoramento do tipo *extract method*.

Palavras-chave: Refatoramento; Teste de Regressão; Ferramentas de Geração de Testes.

Abstract

Refactoring typically require regression testers to verify that the changes applied to the code have preserved the original behavior of the code. Generally, it is difficult to define a set of tests that is effective for this task, since refactoring is not the weight applied in isolated steps. Also, since refactoring issues can be combined with other issues, without code. In this sense, a generation of test cases can contribute to this task by systematically analyzing the code and defining a wide range of test cases that address different constructs. However, a number of studies are not available but are not very effective in detecting faults, particularly refactorings. Based on this, we present two empirical studies that have applied as Randoop and Evosuite tools to generate regression test suites, focusing on the edition of refactoring of the extract method type. Based on the results of the studies, we identified factors that can influence the performance of the tools to effectively test the edition. To validate our findings, we present a set of control models that associate a solution with the ability of the set of tests, of spoken faults related to the refactoring edition. And finally, we present a REFANALYZER, which is a tool that is a solution to what needs to be solved when it is automatically managed, to validate refactoring of the extract method type.

keywords: Refactoring; Regression Test; Test Generation Tools.

Agradecimentos

Agradeço primeiramente a Deus e a Nossa Senhora, que me deram o dom da vida e da saúde.

Aos meus pais, Ivanildo Paulo da Silva e Maria do Socorro Soares Cordeiro e Silva, dois grandes incentivadores da minha história, por todo amor, dedicação, carinho, preocupação, apoio e companheirismo. Nunca mediram esforços para que eu crescesse profissionalmente. Obrigada por, de alguma forma, estarem sempre presente, vibrando com minha felicidade e me dando força em todos os momentos.

Agradeço aos meus irmãos Isailde e Itagive, por todo apoio, amor, carinho e compreensão da ausência. Aos meus sobrinhos Júlio e Maria Vitória, por todo gesto de carinho, pureza e amor verdadeiro. Aos meus cunhados, Josinaldo e Geane, por me apoiar e por todo carinho.

Agradeço em especial ao meu noivo, Jonas Carvalho, por todo amor, compreensão, respeito, apoio, por entender minha ausência e sempre torcer por mim. Ter alguém para caminhar lado a lado, deixa a jornada mais leve e prazerosa. Aos meus sogros e cunhados, por todo carinho, preocupação e respeito.

Meus sinceros agradecimentos aos meus orientadores Everton Leandro G. Alves e Patrícia D. de Lima Machado, que me ensinaram os primeiros passos da pesquisa, sempre com paciência, dedicação, empenho, disponibilidade, respeito e vontade de orientar. Sem eles este trabalho não seria possível. Agradeço também por toda preocupação com minha formação.

Agradeço a Amanda Lima e Luana Gregorio, dois anjos enviados por Deus e que acompanharam toda minha caminhada para realização deste trabalho. Obrigada pelo constante incentivo e por todo carinho.

Agradeço aos meus colegas e companheiros de jornada da sala 105, Alysson, Matheus, Melquisedec, Mirna e Thaciana, por toda troca de conhecimento, pelas conversas, momentos de descontração, carinho e respeito. Agradeço, igualmente, a todos os membros do SPLab.

Aos professores e funcionários da COPIN e do DSC;

A CAPES pelo apoio e suporte financeiro fornecido a este trabalho.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

Capítulo 1

Introdução

Refatoramento consiste na melhoria das estruturas internas do código fonte do *software*, de modo a preservar o comportamento do mesmo [15]. Além disso, é uma prática comum e que faz parte das várias etapas do desenvolvimento ágil. Os principais objetivos do uso de refatoramentos são: reduzir *bed smells* (e.g., duplicação, acoplamento, coesão, complexidade e comprimento) que podem prejudicar a manutenibilidade do código; contribuir para a compreensão do código; ajudar a equipe a repensar e compreender as decisões de *design*; e utilizar elementos de *design* reutilizáveis (padrões de *design*) e módulos de código.[1]

Apesar de já existirem várias ferramentas que dão suporte a aplicação automatizada de edições de refatoramento (e.g., a IDE Eclipse), refatoramento é ainda uma atividade realizada, na prática, manualmente [27; 20]. Por isso, suscetível a erros e consequentemente a introdução de mudanças de comportamento indesejadas [12; 13; 9; 38; 19]. Além disso, na prática, é comum que refatoramentos sejam realizados com outras modificações, podendo aumentar ainda mais as chances de erro [27].

Uma das soluções mais utilizadas para validar refatoramentos recém aplicados, é a utilização de suítes de regressão [20; 27]. Particularmente em contextos ágeis, suites de testes automatizadas são comumente construídas como parte fundamental do processo de desenvolvimento. Tais suites, são utilizadas para aferir o atendimento aos requisitos planejados. Dessa forma, se uma suite de testes for executada com sucesso para a versão base do código fonte, após uma ou mais edições de refatoramento, a suite é reexecutada agora para a versão refatorada. De modo que, ocorrendo falhas, assume-se que o refatoramento não foi

[1] https://www.agilealliance.org/

seguro e que houve mudança de comportamento no sistema. Caso a suite não identifique nenhuma alteração semântica no código, supõe-se que as edições foram aplicadas corretamente e o comportamento do código foi preservado.

Em contrapartida, a medida que as edições de refatoramento são continuamente aplicadas, o custo para reexecutar as suites pode tornar-se inviável. Particularmente, a criação e manutenção de casos de teste podem ser necessárias para lidar com as edições realizadas. Para resolver esse problema, a geração automática de casos de teste parece ser uma abordagem promissora, uma vez que as ferramentas de geração examinam o código sistematicamente em busca de diferentes casos de execução, bem como registrar o comportamento pretendido [38; 26; 35].

O uso de ferramentas de geração para validação de refatoramento é uma prática bastante aplicada na literatura. Soares et al. [38] usa suites de testes geradas pela ferramenta Randoop [29] para validar as edições de refatoramento realizadas por desenvolvedores [37] e por ferramentas que aplicam refatoramento automático. Como resultado, diversas faltas sutís de refatoramentos, foram encontradas e reportadas. Além disso, Mongiovi et al. [26] evoluiu o trabalho de Soares utilizando análise de impacto para orientar a geração de casos de teste.

Apesar do fato de que ambos os estudos mostraram a eficácia dos conjuntos de testes gerados, faz-se necessário de uma investigação mais aprofundada sobre os benefícios e limitações do uso de suites geradas para detectar faltas de refatoramento. Nesse sentido, este trabalho avalia a efetividade de duas ferramentas de geração de testes, Randoop e EvoSuite [16] para a validação de refatoramento, com foco em refatoramentos do tipo *Extract Method* [35]. Um conjunto de estudos empíricos foram realizados, a fim de identificar seus fatores limitadores. Além disso, um conjunto de modelos preditores são propostos, a fim de fornecer a indícios de quando usar tais ferramentas para validar refatoramentos do tipo *extract method*.

1.1 Exemplo Motivante

Em projetos que seguem metodologias ágeis, alterações semânticas de código são constantemente incorporadas ao mesmo. Diante disso, refatoramentos são comuns e necessários para evitar deteorização do código. Porém, diversas vezes, na tentativa de realizar melhoramen-

tos na sintaxe do código, desenvolvedores acabam por introduzir novas faltas em um código previamente estável.

Suponha que, após longas horas de trabalho, um desenvolvedor percebe a oportunidade de reduzir a duplicação de código aplicando um refatoramento do tipo *extract method*. A Figura 1.1 apresenta o código antes e depois do refatoramento. A condição *IF* da linha 3 foi extraída para o método *diffeOfZero* (Figura 1.1-b). Como nenhum erro de compilação ocorreu, o desenvolvedor pode ter a falsa impressão que seu refatoramento foi correto e fazer o *commit* do novo código para o repositório do projeto. Entretanto, o comportanto original do sistema foi alterado. No código original, a condição verifica se a variável `interId` é diferente de zero, onde esta representa um identificador de uma conta. Após a extração, a condição passou a verificar se o *ID* é zero.

Código Fonte 1.1: Método *getAccountByInternalId* a ser refatorado

```
1  public static Account
       getAcountByInternalId(Session
        sess, Integer cid, int
       interId){
2   Account ret = null;
3   if (interId != 0) {
4    ...
5   if(x.size() > 0){
6    ret = (Account)x.get(0);}
7   ...} return ret;}
```

(a) Código original

Código Fonte 1.2: Método *getAccountByInternalId* após a aplicação do refatoramento

```
1  public static Account
       getAcountByInternalId(Session
        sess, Integer cid,int
       interId){
2       Account ret = null;
3           if (diffeOfZero(interId))
                {
4                   ...
5          if(x.size() > 0){
6           ret = (Account)x.get(0);}
7                   ...
8       } return ret;}
9
10 private static boolean
       diffeOfZero(int interId) {
11          return interId == 0;}
```

(b) Código após o refatoramento.

Figura 1.1: Exemplo de refatoramento com falta sutil extraído do projeto JAccounting.

Este é um exemplo de uma falta de refatoramento sutil. Faltas sutis são aquelas onde a semântica do sistema é modificada, porém nenhum erro de compilação é apresentado. Estas faltas passam facilmente despercebidas aos olhos do desenvolvedor, especialmente se este não valida suas edições usando uma suite de regressão, por esta ser ineficiente ou até mesmo inexistente. Neste contexto, suites de regressão geradas automaticamente podem ser de grande utilidade.

1.2 Problema

Em ambientes ágeis, continuamente os projetos são alterados e novas funcionalidades são implementadas e suites geradas, podem ser utilizadas para validar se o comportamento anterior não foi afetado pelas novas alterações, a partir de suites de regressão. Ou seja, aumenta a confiança que o *software* continua estável. Tais suites de regressão, é uma alternativas para validar refatoramentos. Porém, em seu trabalho, Kim *et al.*[31] investigaram o impacto de edições de refatoramento no uso de testes de regressão, utilizando o hitórico de projetos Java. Para isto, avaliaram três projetos *open source* e viram que apenas 22% dos refatoramentos aplicados no estudo, são cobertos pelas suites de testes de regressão do programa. E 62% da suite de testes não exercita os refatoramentos aplicados. Ou seja, na prática a suite pode não ser adequada para testar refatoramentos mesmo tendo uma cobertura de código boa.

Dentro do contexto validar refatoramentos, ferramentas de geração automática de testes, como Randoop e Evosuite, são uma alternativa interessante para projetos de desenvolvimento. Porém, a partir de estudos, verificamos que estas ferramentas possuem fortes limitações e identificamos fatores que podem influenciar o desempenho das ferramentas, quando utilizadas para validar refatoramentos do tipo *Extract Method*. Mediante isto, esse trabalho busca fornecer artefatos para ajudar desenvolvedores a decidir quando o uso de suites geradas é uma opção válida para detecção de faltas de refatoramento.

Assim, o problema geral da nossa pesquisa consiste em como tornar o processo de aplicação de um refatoramento, mais seguro e confiável. De tal modo que, saibamos quando confiar em suites de testes geradas automaticamente.

1.3 Questão de Pesquisa e Objetivos

A fim de guiar nosso trabalho, definimos a seguinte questão de pesquisa:

- **Q1:** Suites de testes geradas por ferramentas de geração automática, são eficazes em detectar faltas de refatoramento?

Além disso, temos os objetivos que nortearam a presente pesquisa, tais como:

1.3.1 Objetivo Geral

Investigar a confiabilidade das suites de testes geradas automaticamente, segundo sua efetividade em detectar faltas de refatoramento, bem como, identificar os fatores que limitam tal aplicabilidade, a fim de aumentar a confiabilidade e segurança do processo de aplicação de um refatoramento.

1.3.2 Objetivos Específicos

- Investigar a eficácia das suites de testes geradas automaticamente, na detecção de faltas de refatoramento;
- Investigar fatores de código que impactam na ineficiência de tal detecção;
- Propor uma abordagem para avaliação da confiabilidade da detecção de faltas de refatoramento, pelas suites de testes geradas automaticamente.

1.4 Contribuições

Resumidamente, as principais contribuições deste trabalho de mestrado são:

- Um conjunto de estudos exploratórios que avaliam a efetividade das ferramentas Randoop e Evosuite, quando validando refatoramento;
- Identificação de um conjunto de características de código que impactam negativamente a efetividade das suites geradas pelo Randoop e Evosuite;

- Um conjunto de modelos de predição da efetividade das ferramentas de geração automática de testes, quanto a detecção de faltas de refatoramento;
- Uma ferramenta que auxilia desenvolvedores a decidir quando confiar em suítes geradas para validar o refatoramento.

1.5 Organização

No capítulo seguinte, apresentamos a fundamentação teórica necessária para o entendimento do nosso trabalho (Capítulo 2). Em seguida, no Capítulo 3 descrevemos os estudos exploratórios, para investigar o uso das ferramentas Randoop e Evosuite no contexto de validação de refatoramento. No Capítulo 4, apresentamos a um novo estudo exploratório, bem como três modelos para predição da efetividade das ferramentas de geração automática de testes, no contexto de validação de refatoramento. No Capítulo 5.1, apresentamos a nossa ferramenta, REFANALYZER. No Capítulo 6, apresentamos os trabalhos relacionados e, por fim, as conclusões no Capítulo 7.

Capítulo 2

Fundamentação Teórica

No presente capítulo, abordaremos conceitos que foram utilizados como base para realização deste trabalho de mestrado. Inicialmente apresentamos conceitos de testes de *software*, com ênfase em teste de regressão. Em seguida, conceituamos refatoramento, onde direcionamos para o tipo *extract method*. Por fim, apresentamos as ferramentas de geração automática de testes de unidade, Randoop e Evosuite.

2.1 Teste de *Software*

A atividade de teste de *software* é realizada com a intenção de encontrar a maior quantidade possível de faltas existentes em um dado programa. Quando esta atividade é realizada, valores como maior confiabilidade e qualidade, são agregados ao *software*. De modo que, maior confiabilidade refere-se à identificação de faltas e sua remoção. Com isto, podemos definir a atividade de teste de *software*, como sendo o processo de executar um programa, com o objetivo de avaliar se ele funciona corretamente de acordo com a especificação [28].

Binder, esclarece três conceitos básicos de testes de *software* [10], tais como:

- **Falta:** consiste na ausência de código no programa ou na presença de código erroneo no mesmo. O que acaba por ocasionar uma falha,
- **Falha:** é o resultado de uma especificação errada ou a a falta de um requisito. Ou seja, é um comportamento diferente do esperado pelo usuário,
- **Erro:** consiste na atividade humana que ocasiona uma falta no sistema.

2.1.1 Testes na Metodologia Ágil

Na década de 1990, várias metodologias começaram a ter visibilidade pública, cada uma com diferentes combinações de fundamentos. Tais metodologias enfatizaram uma estreita colaboração entre a equipe de desenvolvimento e os *stakeholders*, entregas frequentes de valor comercial, pequenas equipes e auto-organizadas, bem como, maneiras inteligentes de criar, confirmar e entregar o código. O termo *"Agile"* foi aplicado a esta coleção de metodologias no início de 2001, quando 17 profissionais de desenvolvimento de *software* se reuniram em Snowbird, Utah para discutir suas idéias compartilhadas e várias abordagens para o desenvolvimento de *software*. Esta coleção de valores e princípios foi expressa no Manifesto para Desenvolvimento de Software Ágil e os doze princípios correspondentes [1].

Segundo a organização Agile Alliance, as metodologias ágeis são subdividas ou classificadas em subgrupos ou áreas de interesse. Tais como: *Extreme Programming*, *Teams*, *Lean*, *Scrum*, *Product Management*, *Devops*, *Design* e *Testing Fundamentals*. O processo de teste em um ambiente ágil também pode se subdivididos em grupos independentes, tais como: *Role-Feature*, *Given-When-Then*, BDD, ATDD, *Acceptance tests*, *Mock Objects*, TDD, *Unit Tests*, *Exploratory testing* e *Usability testing* [2].

Com isso, o processo de teste ágil, pode começar no início do projeto com integração contínua entre desenvolvimento e testes. Teste ágil não é sequencial (no sentido de que é executado somente após a fase de codificação), mas contínuo. A equipe Agile trabalha como uma única equipe em direção a um objetivo comum de alcançar qualidade. Testes ágeis possuem prazos menores (de 1 a 4 semanas), também chamados de iterações. Esta metodologia dá uma melhor previsão sobre os produtos viáveis em curto período de tempo [3].

2.1.2 Tipos de Teste

Testes de *software* podem ser executados em diferentes níveis durante o desenvolvimento do sistema, o que comumente conhecemos por "tipos de testes". Segundo Rocha et al. [23], os principais níveis de testes de *software* são:

[1] https://www.agilealliance.org/agile101/

[2] https://www.agilealliance.org/agile101/subway-map-to-agile-practices/

[3] https://www.guru99.com/agile-testing-a-beginner-s-guide.html

- **Teste de Unidade:** tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. Nesse tipo de teste, o universo alvo são os métodos dos objetos ou mesmo pequenos trechos de código.

- **Teste de Integração:** visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.

- **Teste de Sistema:** avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do *software*. Verificando se o que foi implementado está de acordo com o que os requisitos.

- **Teste de Aceitação:** são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.

2.1.3 Teste de Regressão

Segundo John Wiley et. Al [28], teste de regressão é realizado após uma melhoria funcional ou de uma modificação no código do programa. Seu objetivo é determinar se a mudança realizada alterou outros aspectos do sistema. Comumente, é reexecutado um subconjunto dos casos de teste da suíte ou a suite completa. Realizar teste de regressão é importante, pois as alterações de código e as correções de erros, tendem a ser muito mais propensas a erros do que o código original do programa. Complementarmente, a atividade de teste de regressão deve ser realizada entre duas versões distintas e estáveis do *software*. Proporciona confiança de que as edições recém-introduzidas não interferem nos recursos existentes [41]. Porém, como o *software* está constatemente em alteração, muitos casos de testes são acrescidos à suites, o que acaba por dificultar a execução de todo conjuto de testes. Como forma de solucionar tal problema, Yoo e Harman [41] apontaram três principais linhas. Tais como:

- **Minimização de Casos de Teste:** Detém-se a diminuir o conjunto de casos de testes,

removendo os testes de semânticas redundantes, desnecessários ou obsoletos.

- **Seleção de Casos de Teste:** Seleciona um subconjunto de casos de testes que serão utilizados para testar uma alteração realizada no sistema. Em atividades de testes de regressão, a seleção de casos de testes identifica os casos de testes que são mais relevantes para testar as alterações realizadas entre a versão modificada e a versão anterior do sistema em questão. Diversas abordagens foram propostas para selecionar casos de testes de regressão. Os detalhes do processo de seleção dos testes diferem de acordo com a forma como uma técnica específica define, procura e identifica alterações no programa em teste.

- **Priorização de Casos de Teste:** Determina uma ordem de execução dos testes, a partir da priorização dos testes mais propensos a detectar faltas no programa. Elbaum et al. [14], realizaram um estudo empírico para comparar algumas técnicas de priorização de casos de testes. Os resultados mostraram que todas as técnicas consideradas no estudo melhoram a detecção de faltas da coleção. As técnicas comparadas no estudo incluem estratégias de cobertura de nós ou de métodos, indexação de faltas e diferenças sintáticas entre programas. Concluiram que a melhor técnica à ser utilizada, varia mediante ao que vai ser testado.

Com isso, temos que teste de regressão é uma das principais atividades de teste realizadas durante o ciclo de desenvolvimento de um projeto de *software*, a fim de garantir a manutenção da qualidade do sistema modificado [10]. Além disso, permite aos desenvolvedores efetuarem alterações no código de forma segura, pois se caso alguma falta for introduzidas durante a alteração, esta será descoberta antes da entrega do *software* ao cliente. De modo que ao reexecutar os testes, os desenvolvedores recebe um *feedback* podendo então fazer as correções, como mostra a Figura 2.1

2.2 Refatoramento

Fowler, em seu livro *Refactoring: Improving the Design of Existing Code* [15], define refatoramento como sendo o processo de modificação de um sistema de *software*, de modo que não altere o comportamento externo do código, melhorando sua estrutura interna. É uma

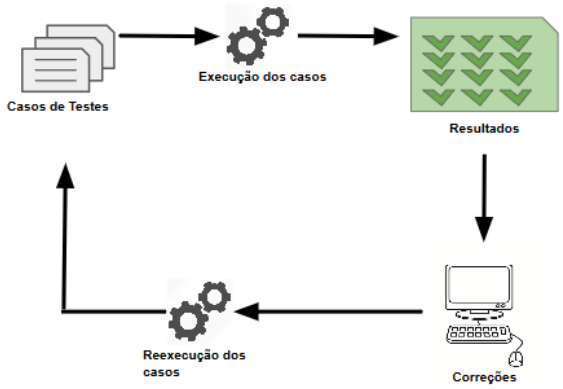

Figura 2.1: Representação visual do procedimento aplicado no Teste de Regressão. Fonte: Autoria própria.

maneira disciplinada de ”limpar”o código, a fim de minimizar as chances de introduzir *bugs*. Em especial, ao realizar um refatoramento, você está melhorando o *design* do código depois de ter sido escrito [15].

Edições de refatoramento normalmente objetivam a melhoria de aspectos não funcionais do sistema (e.g., legibilidade), sem que alterações semânticas sejam introduzidas. Fowler lista em seu catálogo um conjunto de 92 diferentes tipos de refatoramentos[4]. Estes variam em propósito e granularidade. Por exemplo, o refatoramento do tipo *extract method* remove um conjunto de *statements* de um método que pode estar grande, ou pouco legível, para um novo método definido adequadamente. Já refatoramentos do tipo *replace superclass with fields* envolvem elementos de maior granularidade (classes e superclasses).

Na prática, é comum que estas edições sejam executadas manualmente, juntamente com outras modificações, podendo aumentar ainda mais as chances de erros [27]. Martin Fowler cataloga os tipos de refatoramentos, sendo alguns destes: *Extract Method, Move Method, Move Field, Extract Class, Encapsulate Field, Rename Method, Pull Up Method, Pull Up Field, Push Down Method, Push Down Field, Extract Subclass* e *Extract Superclass*[15]. Um dos refatoramentos mais comum é *extract method*. O conceito chave do *extract method* definido por Fowler, consiste em: transformar um fragmento de código em um método cujo nome explica o propósito do método. Ou seja, quando o método está muito complexo ou grande, o programador realiza um refatoramento do tipo *extract method*, a fim de deixar o

[4]http://refactoring.com/catalog/

código mais legível, visualmente menos complexo e que possa ser mantido mais facilmente [27].

Segundo Martin Fowler, para realizar um *extract method* deve-se seguir alguns passos básicos [15], sendo estes:

- Criar um novo método e nomeá-lo mediante a função que terá após a extração;
- Copiar o código que deve ser extraído e colocá-lo no novo método;
- Instanciar as variáveis e parâmetros existentes no escopo para o código extraído;
- Verificar se existe uma variável temporária sendo usada. Caso tenha, deve-se declará-la no método extraído;
- Olhar se existe alguma alteração no escopo devido ao *extract method*;
- Fazer a chamada do método extraído no código original;
- Compilar e testar;

Note que um novo método é criado e o fragmento de código é colocado neste. Outro ponto muito importante, refere-se ao nome dado ao novo método, pois, impreterívelmente, deve-se remeter ao propósito do método. No caso do exemplo fornecido por Fowler [15], tem-se o método original *printOwing*, cujas linhas 6-7 foram refatoradas para o método *printDetails*. Tal nome refere-se a função que o método está executando, neste caso, trata-se de exibir os detalhes da divida, tais como: nome e o montante (Figura 2.2). Murphy Hill et al. [27] realizou um estudo que identificou os tipos de refatoramento mais comumente aplicados por desenvolvedores Java, são eles: *extract method*, *rename method*, *move method* e *pull up method*.

2.3 Ferramenta de Geração Automática de Testes

2.3.1 Randoop

Randoop é uma ferramenta *open-source* para geração automática de suites de teste Java seguindo o padrão do *framework JUnit* [5]. O Randoop gera testes de unidade usando a geração

[5] http://www.junit.org

Código Fonte 2.1: Método de exemplificação do *extract method*, antes da aplicação

```
1 void printOwing(double amount){
2         printBanner();
3
4     //print details
5
6     System.out.println("name: " +
          _name);
7     System.out.println("amount: "
          + amount);
8 }
```

(a) Código original do Extract Method.

Código Fonte 2.2: Método de exemplificação do *extract method*, após a aplicação

```
1 void printOwing(double amount){
2         printBanner();
3     printDetails(amount);
4 }
5
6 void printDetails(double amount){
7     System.out.println("name: " +
          _name);
8     System.out.println("amount: "
          + amount);
9 }
```

(b) Código após o refatoramento.

Figura 2.2: Exemplo de *extract method* fornecido por Fowler em seu livro *Refactoring*.

de teste aleatória dirigida por *feedback*. Esta técnica, gera sequências de invocações de métodos/construtores para as classes em teste. O Randoop executa as sequências que cria, usando os resultados da execução para criar asserções que capturam o comportamento do programa. As suites criadas pelo Randoop podem ser usadas para duas finalidades: encontrar falhas no seu programa, e regressão.

Os testes gerados pelo Randoop focam em pontos do código que possam levar a violações básicas de contrato. Por exemplo, o teste `o1.equals(o2) && o2.equals(o3)` $\rightarrow$ `o1.equals(o3)` verifica se a propriedade transitiva da igualdade de objetos é mantida. Para cada classe a ser testada, o Randoop cria uma sequência de chamadas de métodos e construtores que, por sua vez, criam e alteram os estados dos objetos. Em seguida, todas as sequências criadas são executadas e as saída da execução são utilizadas para criar asserções que capturam o comportamento do sistema.

Recentemente, as suites de teste Randoop têm sido empregadas no contexto da validação de refatoramentos. Ferramentas como SafeRefactor [36; 38] e SafeRefactor Impact [26] usam suites de teste geradas por Randoop para decidir se refatoramentos recém aplicados

foram ou não seguros.

No tocante dos dados de teste, Randoop faz uso de um *pool* de dados composto por valores representativos dos tipos básicos Java (e.g., int, char, boolean, String).

2.3.2 Evosuite

Evosuite é uma ferramenta de geração automática de suites de testes *JUnit* para projetos Java. Dada uma determinada classe, o Evosuite cria sequências de chamadas que maximizam os critérios de teste (e.g. cobertura de linhas e caminhos), enquanto que ao mesmo tempo geram asserções para capturar os comportamentos [7]. O Evosuite, aplica uma abordagem híbrida que gera e otimiza todo o conjunto de testes para satisfazer tais critério de cobertura. Para os conjuntos de teste produzidos, o EvoSuite sugere possíveis oráculos ao adicionar pequenos e efetivos conjuntos de asserções que resumem o comportamento atual. Essas asserções, permitem detectar desvios do comportamento esperado e capturar o comportamento atual para proteger contra defeitos futuros que quebram esse comportamento.[6]

Sendo assim, o Evosuite utiliza um algorítmo evolutivo visando otimizar seu conjunto de casos de testes. Como população, inicia-se com um conjuntos de testes de casos de teste aleatórios e, em seguida, iterativamente, aplica operadores de pesquisa como seleção, mutação e*crossover* para evoluí-los. A evolução é guiada por uma função de aptidão baseada em um critério de cobertura. Uma vez terminada a pesquisa, o conjunto de testes com a maior cobertura de código é minimizado em relação ao critério de cobertura e as afirmações de teste de regressão são adicionadas [18]. Evosuite então verifica para cada teste, se o mesmo está sintaticamente válido, compilando-lo e executando devidademente [34].

2.4 Considerações Finais

No presente capítulo abordamos conceitos primordiais para nossa pesquisa, tais como: Teste de *software*, que é uma atividade cuja intenção é encontrar a maior quantidade possível de faltas existentes em um dado sistema; esclarecemos os três conceitos básicos sobre testes de *software*, que são: falta, falha e erro; e os principais tipos de Teste, tais como: Teste de Unidade, Teste de Integração, Teste de Sistema e Teste de Aceitação. Além disso, conceituamos

[6]http://www.evosuite.org/evosuite/

o Teste de Regressão, onde o objetivo deste é, ao rexecutar um subconjunto dos casos de teste da suíte, determinar se a mudança realizada alterou outros aspectos do sistema. Após isso, falamos sobre Refatoramento, que segundo Fowler é o processo de modificação de um sistema de *software*, de modo que não altere o comportamento externo do código, melhorando sua estrutura interna. Por fim, apresentamos as ferramentas de geração automática de testes de unidade, tais como: Randoop, que gera testes de unidade usando a geração de teste aleatóri dirigida por *feedback*; e Evosuite, que cria sequências de chamadas que maximizam os critérios de teste, enquanto que ao mesmo tempo geram asserções para capturar os comportamentos [7].

Capítulo 3

Estudos Exploratórios

Neste capítulo apresentaremos os estudos exploratórios, realizados com o objetivo de investigar o uso das ferramentas Randoop e Evosuite na detecção de faltas de refatoramento, bem como suas possiveis limitações. Além disso, serão apresentados seus respectivos os resultados, discussões e ameaças à validade.

3.1 Primeiro Estudo Exploratório

Neste estudo, investigamos a eficácia das ferramentas de geração automática de testes, Randoop e Evosuite, quando lidando com dois tipos de faltas de refatoramento.

3.1.1 Objetivos e Questões de Pesquisa

As frequentes edições de refatoramento objetivam reestruturar o código com preservação semântica. A verificação se houve ou não alteração de comportamento, na prática, é realizada combinando edições de refatoramento com a execução de suites de teste de regressão. Entretanto, muitas das vezes, o projeto não dispõe de uma suite de regressão, ou a suite existente não é confiável. Nesse contexto, ferramentas de geração automática de casos de teste podem ser usadas. Porém, até onde conhecemos, não existem na literatura estudos empíricos que analisem a eficácia e/ou limitações dessas ferramentas em gerar casos de teste que detectem faltas de refatoramento.

Pensando nisso, desenvolvemos um estudo empírico, com o intuito de visualizar os

possíveis benefícios e limitações das ferramentas de geração de casos de teste, quando lidando com faltas de refatoramento. Neste estudo, focamos em duas das ferramentas de geração mais usadas na prática, Randoop (versão 3.0.1)[1] e Evosuite (versão 1.0.3) [2]. Tais ferramentas foram escolhidas dado ao fato que suites de teste geradas por estas, têm sido usadas para validar mudanças de código em diversos trabalhos [36; 38; 26; 17; 33].

A fim de guiar nosso estudo, duas questões de pesquisa foram definidas:

Q1: Um desenvolvedor pode confiar nas suites de regressão geradas para detectar faltas de refatoramento?

Q2: Quais fatores limitam a eficiência das ferramentas de geração?

3.1.2 Objetos e Configuração

Para este experimento, foram selecionados três sistemas *open-source* Java, nos quais consideramos representativos para o mundo real. Estes são os objetos experimentais escolhidos para o nosso estudo:

- HealthCard ($\approx$ 2KLOC)[3], aplicação que gerencia consultas médicas em *smart cards*;
- JAccounting ($\approx$ 7KLOC)[4], sistema de contabilidade e *Web Spider Engine*; e
- Jmock ($\approx$ 5KLOC)[5], biblioteca popular que auxilia a criação de testes de unidade usando objetos mock.

É válido ressaltar que tanto o HealthCard, quanto JAccounting, foram utilizados como objetos em outras pesquisas científicas [32; 24]. Quanto ao Jmock, trata-se de um projeto consideravelmente estável, além de ser uma biblioteca bastante utilizada. Outra ressalva, é que dentre os objetos escolhidos, apenas o JMock possui uma suite regressão, cujos testes foram criados manualmente, acoplada ao projeto. A fim de sistematizar a condução do nosso

[1] http://randoop.github.io/randoop/
[2] http://www.evosuite.org/
[3] https://sourceforge.net/projects/healthcard/
[4] https://jaccounting.dev.java.net/
[5] http://www.jmock.org/

experimento, optamos por gerar, utilizando as ferramentas Randoop e Evosuite, cinco suites regressão para cada objeto experimental, totalizando trinta. Estas suites refletem o comportamento de uma versão estável de cada sistema. As trinta suites foram geradas seguindo a mesma configuração de tempo (150 segundos) e 500 como número máximo de testes por arquivo. Uma ressalva sobre a escolha do tempo escolhido, este parâmetro de entrada de 150 segundos, não correspondeu ao real tempo gasto. Pois na prática, especialmente para a ferramenta Evosuite, após o processo de geração dos testes, existe processo de minimização da suite gerada que, em média, gastou de duas a três horas.

Todas as gerações foram realizadas em um computador Desktop com processador de 3.10GHz, memória RAM de 8 GB, sistema operacional Windows 10. Usando as configurações mencionadas anteriormente, optamos por utilizar a média das taxas de cobertura das replicações para cada ferramenta, devido a similaridade entre estas taxas. Não houve nenhum caso em que o valor da cobertura, foi consideravelmente diferente. Com isso, presumimos a partir dessa similaridade, que as suites cobriram os mesmos elementos de código. E, descartamos uma análise mais detalhada, para validar tal afirmação. As referidas médias estão dispostas nas Tabelas 3.1 e Tabela 3.2, respectivamente. A análise de cobertura foi realizada usando a ferramenta *EclEmma* [6]. As versões usadas como unidades experimentais, bem como as suites de teste geradas estão disponíveis no nosso site[7].

Tabela 3.1: Média da coberturas das replicações Randoop por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
HealthCard	85,70%	78,00%
JAccounting	48,86%	33,50%
JMock	17,40%	34,00%

[6]http://www.eclemma.org/

[7]https://sites.google.com/copin.ufcg.edu.br/automaticgeneratetests/

Tabela 3.2: Média da coberturas das replicações Evosuite por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
HealthCard	91,32%	91,78%
JAccounting	33,12%	30,08%
JMock	18,92%	46,68%

3.1.3 Faltas Injetadas

Como nosso objetivo de investigação, nesse primeiro experimento, é analisar de forma controlada o comportamento das ferramentas Randoop e Evosuite quando lidando com faltas de refatoramento, decidimos focar nosso experimento em dois tipos de faltas, *Troca Condição por Valor Fixo* e *Troca Operador da Condição*. Estes tipos de faltas, no contexto do nosso estudo, estão associados a refatoramento do tipo *extract method*. Escolhemos esses dois tipos de faltas, pois emulam situações onde um desenvolvedor realiza a extração da condição de um comando *"IF"* para um novo método e, por descuido, acaba modificando a semântica da condição no método de destino. Ou seja, ao realizar um *extract method*, o desenvolvedor altera a semântica do sistema. Variações dessas faltas são bastante comuns na prática, visto que a mecânica estabelecida por Fowler para o *extract method* [15] combina diversas mini-modificações. Dentre essas modificações, Fowler recomenda que o desenvolvedor crie primeiro a estrutura básica do novo método (código estritamente necessário para que não haja erros de compilação) e somente depois atualize o corpo desse método com o código extraído. Devido a circunstâncias variadas (e.g, restrições de tempo), desenvolvedores podem acabar esquecendo de atualizar o novo método, introduzindo assim um falta. Vale salientar que faltas semelhantes têm sido usadas em outros estudos empíricos que lidam com problemas relacionados a refatoramentos [4; 2]. Alves *et al.* trataram com o mesmo procedimento de inserção de faltas, porém diferindo do tipo de refatoramento utilizado, neste caso o *move method.*

A fim de ilustrar a falta *Troca Condição por Valor Fixo*, suponha o código extraído do sistema HealthCard (Figura 3.1). Para injetarmos tal falta, a condição do *"IF"* (linha 2) é extraída para o método `verifyNullObjects`. Porém, no novo método,

ao invés da reprodução da condição original (`o1 == null || o2 == null`), encontramos o retorno direto do valor `false` (falta). Assim, apesar de não incluir nenhum erro de compilação, a modificação faz com que o corpo do comando *"IF"* em `verifyNullObjects` nunca seja executado, o que é uma clara alteração do comportamento original do método.

Código Fonte 3.1: Método de exemplificação da falta do tipo *Troca condição por Valor Fixo*, antes da aplicação

```
1  boolean areEqual(Object o1,
       Object o2 ){
2   if (o1 == null || o2 == null) {
3    return o1 == null && o2 == null
         ;
4   } else if (isArray(o1)) {
5    return isArray(o2) &&
         areArraysEqual(o1, o2);
6   } else {
7    return o1.equals(o2);
8   }}
```

(a) Código original.

Código Fonte 3.2: Método de exemplificação da falta do tipo *Troca condição por Valor Fixo*, após da aplicação

```
1  boolean areEqual(Object o1,
       Object o2 ) {
2   if (verifyNullObjects(o1, o2)) {
3    return o1 == null && o2 == null
         ;
4   } else if (isArray(o1)) {
5    return isArray(o2) &&
         areArraysEqual(o1, o2);
6   } else {
7    return o1.equals(o2);
8   }}
9  boolean verifyNullObjects(Object
       o1, Object o2) {
10   return false;}
```

(b)Código com falta injetada.

Figura 3.1: Exemplo de falta do primeiro tipo *Troca Condição por Valor Fixo* injetada quando realizando um *extract method*.

Para ilustrar a falta *Troca Operador da Condição*, temos o código também extraído do sistema HealthCard (Figura 3.2). Para injetarmos a falta, a condição do *"IF"* (linha 4) é extraída para o método `checkNullity`. Porém, no novo método, ao invés da reprodução da condição original (`array[i] != null`), encontramos a inversão do operador da condição `array[i] == null` (falta). Assim, apesar de não incluir ne-

nhum erro de compilação, a modificação faz com que o corpo do comando *"IF"* em `countNotNullObjects` seja executado com uma semântica totalmente diferente, a medida que a condição deveria verificar se os objetos do `Array` não são `Null`, ele verifica os que são `null`. O que é uma clara alteração do comportamento original do método.

Código Fonte 3.3: Método de exemplificação da falta do tipo *Troca Operador da Condição*, antes da aplicação

```
1 public short countNotNullObjects(
      Object[] array){
2  short count = (short) 0;
3  for (short i = (short) 0; i < (
       short) array.length; i++){
4   if (array[i] != null) {
5    count += 1;}
6   }return count;
7 }
```

(a) Código original.

Código Fonte 3.4: Método de exemplificação da falta do tipo *Troca Operador da Condição*, após a aplicação

```
1 public short countNotNullObjects(
      Object[] array){
2  short count = (short) 0;
3  for (short i = (short) 0; i < (
       short) array.length; i++){
4   if (checkNullity(array, i)) {
5    count += 1;}
6   }return count;
7 }
8 private boolean checkNullity(
      Object[] array, short i) {
9  return array[i] == null;}
```

(b)Código com falta injetada.

Figura 3.2: Exemplo do falta *Troca Operador da Condição* injetada quando realizado um *extract method*.

3.1.4 Procedimento de Inserção das Faltas

Dadas as unidades experimentais e os tipos de faltas injetadas, nosso estudo foi executado de acordo com o seguinte procedimento: para cada sistema, 40 versões faltosas foram criadas. Dentre estas, 20 são referentes a falta do tipo *Troca Condição por Valor Fixo* e 20 do tipo *Troca Operador da Condição*, totalizando 120 faltas injetadas no experimento. É válido ressaltar que, cada falta foi inserida manualmente e, cada versão do código inclui uma única falta por vez, podendo ser tanto do tipo *Troca Condição por Valor Fixo* quanto do tipo *Troca Operador da Condição* em ambos os casos, sempre associada a uma edição de refatoramento

do tipo *extract method*. As faltas do primeiro tipo foram injetadas seguindo o procedimento descrito na Tabela 3.1.4. Enquanto as faltas do tipo *Troca Operador da Condição* foram injetadas seguindo o procedimento descrito na Tabela 3.1.4. Além disso, todas as faltas foram inseridas em métodos públicos e, não públicos passíveis de detecção. Ou seja, que são chamados diretamente ou indiretamente por métodos de visibilidade pública. Dessa forma, asseguramos que as faltas são alcançáveis pelas ferramentas.

Procedimento da falta *Troca Condição por Valor Fixo*.

Sendo S o conjunto de métodos de um determinado sistema: 1. Selecionar randomicamente um método m de S. Em seu corpo deve existir um comando IF com uma condição qualquer c; 2. Extrair c para um novo método. Este novo método vai possuir um nome a ser selecionado de um *pool* de nomes pré-estabelecido; 3. Se a extração realizada em 2 gerar algum erro de compilação, reverter a extração e voltar para o passo 1; 4. Inserir a falta alterando a condição c no novo método para o valor $FALSE$; 5. Remover m de S.

No total, nosso estudo lida com 120 versões faltosas. Sendo 60 do tipo *Troca Condição por Valor Fixo* e 60 *Troca Operador da Condição* (20 faltas para cada um dos 3 sistemas). Para cada versão faltosa, 5 replicações da ferramenta Randoop e 5 da Evosuite, foram executadas e métricas relacionadas a detecção foram coletadas, bem como aspectos subjetivos foram analisados. Além disso, para os casos onde as replicações não detectaram alguma falta injetada, as validamos manualmente criando casos de testes que as detectavam. Ou seja, criamos manualmente casos de testes para estes. Ao executar o processo de inserção das faltas, nós coletamos, em cada execução, o número de casos de teste que falham e/ou apresentaram erro.

Para o nosso contexto de detecção de faltas de refatoramento, não separamos falhas de erro. Ao analisarmos os casos onde foram lançados apenas erro, vimos que, quando um

Procedimento da falta *Troca Operador da Condição*.

Sendo S o conjunto de métodos de um determinado sistema:
1. Selecionar randomicamente um método m de S. Em seu corpo deve existir um comando IF com uma condição qualquer c;
2. Extrair c para um novo método. Este novo método vai possuir um nome a ser selecionado de um *pool* de nomes pré-estabelecido;
3. Se a extração realizada em 2 gerar algum erro de compilação, reverter a extração e voltar para o passo 1;
4. Inserir a falta alterando o operador da condição de: "=="para "!=", "!="para "==", $\gg$"para $\ll$", $\ll$"para $\gg$", $\gg$="para $\ll$="ou $\ll$="para $\gg$=". Caso a condição extraída tenha mais de um operador, todos são alterados;
5. Remover m de S.

caso de teste apresenta um erro, o mesmo está diretamente ou indiretamente relacionado com a falta inserida no código. Diante disso, nós tratamos detecção, apenas quando um caso de teste falha ou gera um erro. Enquanto não-detecção, nós entendemos como sendo uma suite de testes que não apresentou nenhum dos dois, ou seja, a suite continuou com todos os casos de testes passando. Além disso, consideramos que uma falta foi detectada quando pelo menos um caso de teste, dentre as cinco suites geradas pela ferramenta em questão, falhar.

A Figura 3.3 sumariza a configuração do nosso estudo para um dado objeto experimental (sistema).

3.1.5 Resultados e Discussão do Primeiro Experimento

A Figura 3.4 apresenta uma visão geral da taxa de detecção das faltas injetadas no contexto do nosso estudo exploratório. Das 120 faltas injetadas, 68,3% foram detectadas pelas suites geradas por ambas ferramentas. A fim de melhor organizar a análise e discussão dos resultados do estudo, decidimos por dividi-los em dois grupos: i) quando as suites conseguiram detectar as faltas injetadas e ii) quando não houve detecção.

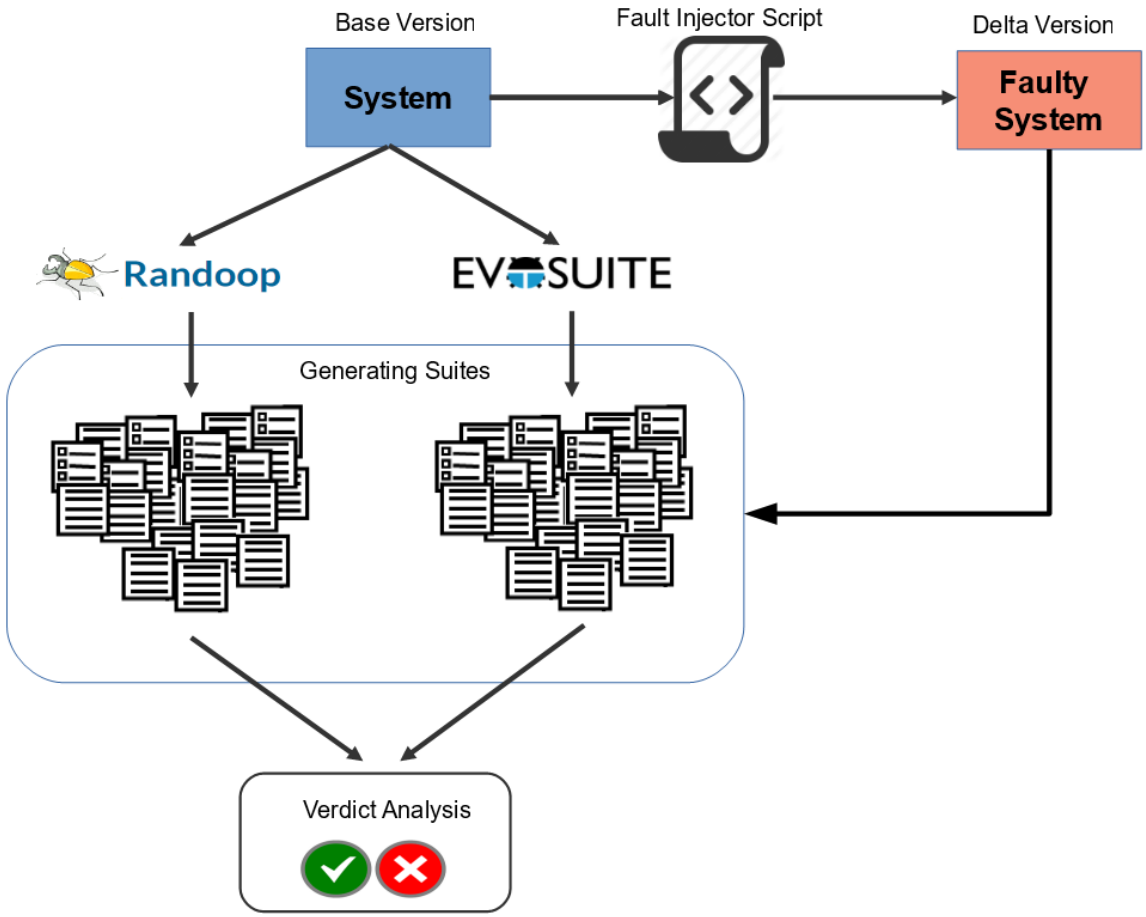

Figura 3.3: Representação visual do procedimento aplicado no estudo empírico para cada um dos sistemas. Fonte: Autoria própria.

3.1.6 Quando houve detecção

As seções seguintes apresentam resultados de quando houveram detecções por ferramenta, por falta e quando houve detecção simutânea para ambas ferramentas.

Detecção por Ferramenta

No contexto do nosso estudo, definimos que uma falta é detectada quando a suite JUnit gerada pela ferramenta apresenta casos de teste que falham ou geram erros.

Fazendo um levantamento geral (Figura 3.5) das detecções por ferramentas, Randoop ou Evosuite, temos, respectivamente, 50,8% e 64,2%, de faltas detectadas. Fazendo uma separação por tipo de falta, temos que do tipo *Troca Condição por Valor Fixo* das 120 faltas, as suites Randoop detectaram 6 do JAccounting, 11 do JMock e 15 do HealthCard. Enquanto

Figura 3.4: Proporção de faltas detectadas. Fonte: Autoria própria.

para Evosuite foram 8 do JAccounting, 11 do JMock e 19 do HealthCard. No casos da falta do tipo *Troca Operador da Condição*, as suites Randoop detectaram 5 do JAccounting, 8 do JMock e 16 do HealthCard. Enquanto as Evosuite detectaram 7 do JAccounting, 13 do JMock e 19 do HealthCard. Com isso, podemos afimar que, independente do tipo de falta, ambas ferramentas não obtiveram desempenho safisfatório, visto que em todo o experimento estas detectaram menos de 65% da quantidade total de faltas injetadas.

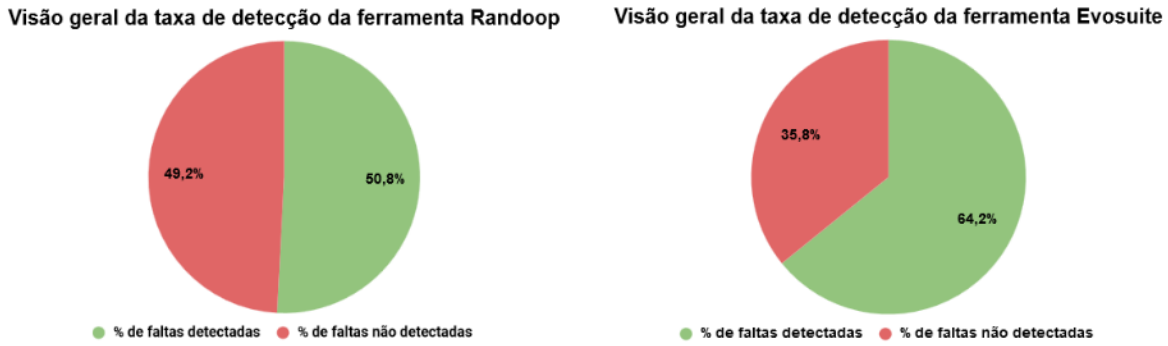

Figura 3.5: Proporção de faltas detectadas pelas ferramantas Randoop e Evosuite, respectivamente. Fonte: Autoria própria.

Outra observação a ser feita é quanto à diferença nas taxas de detecção para as ferramentas. Ambas utilizam mecanismos aleatórios para geração de casos de teste. Porém, a

Tabela 3.3: Teste de proporção

Proporção por Ferramenta *p-value*	**Proporção por Tipo de falta *p-value***
7.722e-07 %	0.0584 %

primeira realiza testes caixa preta e a segunda caixa branca. O algoritmo de geração do Randoop portanto, é limitado a itens públicos e, ao tentar construir parâmetros que não correspondem ao conjunto predito, a ferramenta não consegue gerar um caso de teste, aborta, e passa para o próximo método. Isto foi um ponto que influenciou fortemente na sua taxa de detecção. Quando observamos o Evosuite, a taxa foi um pouco melhor, apesar das limitações encontradas para o Randoop serem comuns ao Evosuite. Isso foi possivel pois o algoritmo de geração do Evosuite cria suites de testes simplificadas, mas que cobrem a maior quantidade de código possivel.

Além disso, pudemos notar que as maiores taxas de detecções alcançadas estão relacionadas com o sistema cujas replicações obtiveram, em média, as maiores coberturas de caminhos em ambas ferramentas, no caso do Randoop (HealthCard - 78,00%) e para o Evosuite (HealthCard - 91,78%). Porém, mesmo para o HealthCard, houveram faltas de ambos os tipos que não foram detectadas. Mais precisamente, para a falta *Troca Condição por Valor Fixo*, a média de detecção para o HealthCard foi de (85%) e para *Troca Operador da Condição* foi (87,5%). Isso mostra que alguns casos passaram despercebidos de detecção, acarretando na permanencia de *bugs* na versão do sistema.

Detecção por Tipo de falta

Sabendo-se que 20 faltas, para cada tipo, foram injetadas em cada unidade experimental, temos para a falta do tipo *Troca Condição por Valor Fixo*, o número de faltas que foram detectadas, sendo: 9 do JAccounting, 15 do JMock e 19 do HealthCard. No caso da falta do tipo *Troca Operador da Condição*, temos repectivamente: 7 do JAccounting, 13 do JMock e

19 do HealthCard. Isso nos dá, respectivamente, uma taxa de 71,7% e 65,0% como mostra a Figura 3.6. Diante disso, aplicamos o teste de proporção, com 95% de confiança (Tabela 3.3). A partir desse testes, é possível afirmar que o tipo da falta não gera diferenças estatísticas para os resultados das ferramentas. Apesar disso, numericamente, as ferramentas detectaram mais faltas do tipo *Troca Condição por Valor Fixo*, como mostra a tabela de detecção por tipo de falta 3.4.

Tabela 3.4: Detecção por tipo de falta

Sistema	**Troca Condição por Valor Fixo**	**Troca Operador da Condição**
HealthCard	95%	95%
JAccounting	45%	35%
JMock	75%	65%

Uma análise interessante a ser feita, considera a taxa de casos de teste que falham para cada tipo de falta injetada, *Troca Condição por Valor Fixo* e *Troca Operador da Condição*, respectivamente. Considerando que foram detectadas para cada tipo, respectivamente, 43 e 39, e sabendo que apenas uma falta foi injetada por vez, em média, 71,7% e 65,0% dessas faltas foram detectadas. Existindo casos onde 570 (2,49%) dos casos de teste falharam devido a uma única falta. Esse elevado número de falhas deve-se ao fato do Randoop criar casos de testes muito semelhantes entre si. Em contrapartida, apesar do número de casos de testes gerados pelo Evosuite ser consideravelmente menor, quando comparado com a ferramenta anterior, houveram casos onde, para a mesma falta 5 casos (1,16%) de uma suite falharam. Por isso, a tendência é que, quando uma falta é detectada, esta, na maioria das vezes, leva diversos casos de teste a falharem.

Em alguns contextos, o elevado número de falhas pode ser visto como algo positivo, por reafirmar a presença de uma falta no código, aumentando a confiança que os testes são de fato válidos. Por outro lado, pensando no uso das suites como validadores de mudanças de refatoramento, uma grande quantidade de casos de teste que falham pode dificultar a análise

e localização da falta. Principalmente no contexto ágil, onde refatoramentos são frequentes mas recursos para análise e depuração dos testes são limitados, analisar um grande número de casos de teste que falharam para tentar entender e localizar uma falta pode ser algo custoso, tedioso e/ou não viável na prática.

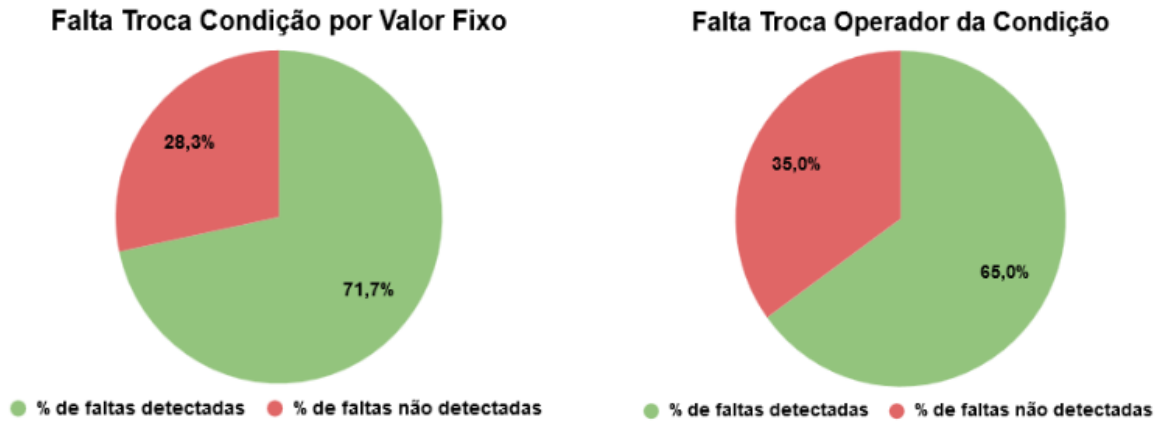

Figura 3.6: Proporção detecção por tipo de falta. Fonte: Autoria própria.

3.1.7 Quando não houve detecção

Dentre as 120 faltas injetadas, 31,7% não foram detectadas (Figura 3.4). Número este referente as suites do Randoop e Evosuite. No tocante a não detecção, podemos observar que para o primeiro tipo de falta, o JAccounting teve das 20 faltas injetadas 11 (55%) que não detectaram, para o Jmock esse número também foi significativamente alto, sendo 5 (25%) e no caso do Healthcard 1 (5%). Para o segundo tipo de falta, o JAccounting obteve das 20 faltas injetadas 13 (65%) que não detectaram, para o Jmock esse número também foi significativamente alto, sendo 7 (35%) e no caso do Healthcard 1 (5%).

A fim de garantir que as faltas não detectadas pelas suites Randoop e Evosuite são de fato passíveis de detecção, foram criados manualmente casos de teste que detectaram as faltas que não foram detectadas em nenhuma replicação. Como exemplo, temos o método *areArrayElementsEqual* do sistema *JMock*, cuja visibilidade é não pública e possui como parâmetros de entrada e como retorno, parâmetros de tipos primitivos. A falta inserida foi a *Troca Condição por Valor Fixo* e nenhuma das suites geradas pelas ferramentas Randoop e Evosuite, conseguiram detectar a referida falta. A Figura 3.7 mostra um caso de teste escrito manualmente, que detecta a falta injetada. Note que o caso de teste refere-se ao método *eval*,

em que este possui visibilidade pública e consegue acessar indiretamente *areArrayElementsEqual*, cuja falta foi inserida.

Código Fonte 3.5: Caso de teste criado manualmente que detecta a falta inserida

```
1  public void testComparesTheElementsOfAnArrayOfPrimitiveTypes(){
2          int[] i1 = new int[]{1, 2};
3          int[] i2 = new int[]{1, 2};
4          int[] i3 = new int[]{3, 4};
5          int[] i4 = new int[]{1, 2, 3, 4};
6
7          Constraint c = new IsEqual(i1);
8          assertTrue("Should equal itself", c.eval(i1));
9          assertTrue("Should equal a similar array", c.eval(i2));
10         assertTrue("Should not equal a different array", !c.eval(i3));
11         assertTrue("Should not equal a different sized array", !c.eval(i4
               ));}
```

Figura 3.7: Teste manual que identificou a falta inserida no refatoramento.

Devido ao grande número de faltas não detectadas, podemos responder **Q1** e afirmar que as suites não são efetivas para detectar faltas de refatoramento e os desenvolvedores confiassem unicamente nas suites geradas automaticamente. Pois, das 60 faltas do tipo *Troca Condição por Valor Fixo* 17 passaram sem detecção e para as 60 faltas do tipo *Troca Operador da Condição* 21 passam sem detecção, sumarizando podemos dizer que ($\approx$ 1/3) das faltas não foram descobertas.

Uma justificativa direta para a ineficiência das suites do JAccouting e JMock são as baixas taxas de cobertura para ambas ferramentas utilizadas. A abordagem usada pelas ferramentas desconsideram a geração de testes para métodos não públicos e/ou que recebem objetos complexos como parâmetro, o que levou a uma baixa taxa de cobertura geral. Em uma análise rápida, visualizamos tal problemática no tocante a métodos não públicos. Para cada um dos sistemas, escolhemos aleatoriamente 10 classes e contamos quantos métodos são públicos e não públicos. Os resultados dessa análise mostraram que 100% dos métodos analisados foram públicos para o HealthCard, enquanto essa taxa cai para 87% para o JAccounting e 61% para o JMock. Tais resultados são condizentes com as taxas de cobertura atingidas pela Randoop (quanto mais métodos públicos maior a cobertura das suites geradas), e inversamente

proporcionais as taxas de faltas não detectadas (quanto mais métodos não públicos menor a taxa de faltas detectadas). Tal limitação pode ser crucial na prática, visto que muitos dos refatoramentos, visando melhorar a legibilidade e facilitar futuras manutenções do código, introduzem elementos (e.g., métodos) de visibilidade não pública (e.g., refatoramento do tipo *hide method*). Faltas de refatoramento podem estar relacionados a tais elementos e, como estes não são considerados durante a geração das ferramentas, essas faltas tendem a passar desapercebidas ou não conseguem ser reproduzidas.

Com relação aos métodos que recebem como argumento de entrada e saída objetos complexos (entenda como objetos complexos aqueles que não são de tipos primitivos), ambas as ferramentas tentam criar ou reproduzir um objeto válido, não obtendo êxito, o procedimento é abortado e um outro método é escolhido para a geração. A Figura 3.8 exemplifica o método *reportIncompatibleCheckedException*, retirado do sistema JMock, que não foi testado pelas suites geradas, por receber um objeto complexo como parâmetro. Ambas as ferramentas não conseguiram criar e/ou configurar o parâmetro de entrada do tipo *Class[]* e o de retorno do tipo *void* para usar nos casos de teste, deixando esse método fora da geração. É válido ressaltar que o referido método é chamado indiretamente por um métodos de visibilidade pública e a falta inserida neste, foi a *Troca Operador da Condição*. A Figura 3.9 trata-se de uma caso de teste retirado da suite de regressão fornecida pelo projeto do *JMock*, onde o referido testes detectou a falta inserida no método *reportIncompatibleCheckedException* (Figura 3.8).

Código Fonte 3.6: Método que não foi coberto pelas ferramentas de geração de teste, Randoop e Evosuite

```
1 private void reportIncompatibleCheckedException(Class[] allowedTypes) {
2  StringBuffer message = new StringBuffer();
3  ...
4  if (allowedTypes.length == 0) {...}
5  ...
6  fail(message.toString());}
```

Figura 3.8: Método negligenciado pela geração das ferramentas.

Dessa forma, podemos responder **Q2** e afirmar que as suites geradas pelas ferramentas Randoop e Evosuite são bastante limitadas pela quantidade de métodos não públicos e pela

Código Fonte 3.7: Caso de testes manual, que detectou a falta inserida.

```
1  public void
       testThrowsAssertionFailedErrorIfTriesToThrowIncompatibleCheckedException
       () throws Throwable{
2   Class[] expectedExceptionTypes = {ExpectedExceptionType1.class,
        ExpectedExceptionType2.class};
3   Invocation incompatibleInvocation = new Invocation("INVOKED-OBJECT",
        methodFactory.newMethod("methodName", MethodFactory.NO_ARGUMENTS,
        void.class, expectedExceptionTypes), null);
4   try {
5    throwStub.invoke(incompatibleInvocation);
6   }catch (AssertionFailedError ex) {
7    String message = ex.getMessage();
8    for (int i = 0; i < expectedExceptionTypes.length; i++) {
9    AssertMo.assertIncludes("should include name of expected exception
         types", expectedExceptionTypes[i].getName(), message);}
10   AssertMo.assertIncludes("should include name of thrown exception type",
         THROWABLE.getClass().getName(), message);
11   return;}
12  fail("should have failed");}
```

Figura 3.9: Caso de teste que detectou a falta inserida no método *reportIncompatibleCheckedException*.

complexidade dos parâmetros recebidos e retornados dos métodos. Tais limitações impactam diretamente no poder de detecção dessas suites, pois estes elementos de código (métodos não públicos e objetos complexos) são elementos básicos de qualquer sistema orientado a objetos.

Mais uma vez, vale destacar que faltas injetadas em métodos não públicos podem ser detectadas indiretamente se métodos públicos as chamam e são usadas durante a geração. Porém, nossos resultados mostram que tal situação é rara. Por exemplo, para as 60 faltas injetadas do tipo *Troca Condição por Valor Fixo* foram 11 em métodos não-públicos. Apenas uma dessas faltas foi detectada.

Em alguns pouco casos, apesar da falta ter sido injetada em métodos não públicos, houve detecção para ambos os tipo de faltas. Ao analisarmos esses casos separadamente, percebe-

mos que tais detecções ocorreram pelo fato de que os métodos onde as faltas foram injetadas são invocados por outros métodos públicos (*public*) e esses sim fizeram parte das sequências de chamadas dos casos de teste gerados. Assim, para esses casos, a detecção ocorreu de forma indireta. Tal fato pode dificultar ainda mais o uso desses casos de teste para compreensão e localização de faltas, visto que a tarefa de rastrear uma falta torna-se mais simples quando os métodos são invocados diretamente nos casos de teste [2].

3.1.8 Análise Combinada

Dentro do contexto de faltas detectadas, houveram casos onde ambas ferramentas conseguiram detectar as mesmas faltas. Para a falta do tipo *Troca Condição por Valor Fixo*: 45% do JAccounting, 75% do JMock e 95% do HealthCard, dos casos foram identificados simutâneamente tanto pela ferramenta Randoop, quanto pela Evosuite. No caso da falta do tipo *Troca Operador da Condição*: 35% do JAccounting, 65% do JMock e 95% do HealthCard, dos casos foram identeficados simutâneamente. Isso nos dá respectivamente uma média de 71% e 65% dos casos que foram detectados por ambas as ferramentas, número este relativamente baixo.

Quando analisamos com cuidado os casos onde as faltas foram detectadas por ambas ferramentas, podemos observar que estes se referem basicamente a casos onde as faltas se localizam em métodos públicos (*public*) e que lidam com tipos de dados simplificados (e.g., tipos primitivos, tipos estruturados simples).

Por fim, houveram casos onde a falta foi detectada por uma única ferramenta e a outra não. Para os casos que foram indentificados pelo Evosuite e que o Randoop não conseguiu detectar, temos para cada sistema: JAccounting 5 casos, JMock 9 casos e Healthcard 7 casos. Quando analisamos de forma inversa, ou seja, os casos que foram indentificados pela ferramenta Randoop e que o Evosuite não conseguiu detectar, temos que: JAccounting 1 caso, JMock 4 casos e apenas para o Healthcard que não teve nenhum caso para este cenário. Diante disso, note que pode não ser muito confiável utilizar tais ferramentas de forma separadas, pois muitas faltas deixaram de ser detectadas. Com isto, usá-las em conjunto, pode ser uma boa alternativa.

3.1.9 Ameaças à Validade

Validade Interna

Outras métricas, além das taxas de detecção, poderiam ter sido utilizadas, tais como: cobertura de elementos de código impactados. No entanto, nosso estudo teve como objetivo analisar a eficiência das suites geradas em ajudar a detectar faltas de refatoramento. Neste contexto, os desenvolvedores muitas vezes procuram um veredicto simples para decidir se a sua edição é segura. Outro ponto seria a investigação de outras ferramentas de geração (por exemplo, JCrasher), além de Randoop e Evosuite. No entanto, estas são as duas ferramentas mais utilizados na prática e foram usadas para validar as edições de refatoramento em outros trabalhos [26; 38]. Como trabalho futuro, planejamos expandir nosso estudo fazendo uso de outras ferramentas. Outra limitação do nosso trabalho, refere-se à quantidade de replicações utilizadas no estudo, cinco replicações. Como trabalho futuro, queremos expandir todo o estudo para um número maior de replicações.

Validade Externa

Em termos de validade externa, as conclusões do nosso estudo não se generalizam além do seu escopo (sistemas escolhidos e tipos de faltas de refatoramento). No entanto, acreditamos que os sistemas utilizados são bons representantes. Todos os três são sistemas *open source*, que também foram usados em outros estudos empíricos [4; 25]. Da mesma forma, apenas dois tipos de faltas de refatoramento foram injetadas, ambas relacionadas a um único tipo de edição de refatoramento (*extract method*). Novamente, essas faltas foram inspiradas em faltas usadas em estudos relacionados à validação de refatoramento [4; 3]. Finalmente, o refatoramento *extract method* é conhecido por ser uma dos mais utilizados na prática [27].

Validade de Construção

Em termos de validade de construção, ambas as ferramentas sob investigação, usam aspectos aleatórios para gerar suas suítes. Assim, uma geração diferente poderia criar conjuntos com diferentes resultados. Para minimizar essa ameaça, o estudo foi replicado cinco vezes e os resultados relatados consideram os resultados médios.

Validade de Conclusão

Todos os resultados aqui mencionados e conclusões tomadas, são válidos para as unidades experimentais, ferramentas de geração de testes e a metodologia utilizada.

3.2 Segundo Estudo Exploratório

Durante a realização do nosso primeiro estudo experimental as ferramentas de geração automática de testes, Randoop e Evosuite, lançaram novas versões, com melhorias agregadas à estas. Especificamente para o Evosuite, tais melhorias referem-se a integração do acesso de *API private* (PA) e o *functional mocking* (FM) para a geração de teste de unidade baseada em pesquisa [8]. Quanto ao Randoop, as mudanças estão relacionadas à correção e melhoria de *bugs* existentes e, além disso, o tratamento de *flaky test*, testes que se comportam de forma diferente em diferentes execuções. Quando levantamos a hipótese de que estes novos ajustes poderiam impactar diretamente no poder de detecção das ferramentas, dentro do contexto de validar refatoramentos, optamos por replicar o estudo descrito na Seção 3. Onde neste, validaríamos se as conclusões estabelecidas usando as versões anteriores das ferramentas, permaneceriam válidas para as novas versões.

Diante disso, realizamos a replicação do primeiro estudo (Estudo 3), diferindo em alguns fatores, tais como: as versões das ferramentas Randoop (versão 3.1.5) e Evosuite (versão 1.0.5) e a adição do comando *'–ignore-flaky-tests==true'* na configuração do Randoop. Isso foi necessário pois, ao tentar gerar as novas suites de teste, o próprio Randoop não executou por completo o processo de geração e reportou o erro informando que havia identificado *flaky test*. Dessa forma, utilizamos a opção *'TRUE'*, para ignorar tais testes.

Toda a configuração utilizada no primeiro estudo 3.1.1, foi mantida. É importante ressaltar que, todas as gerações foram realizadas no mesmo computador e seguindo o mesmo procedimento e configuração. As coberturas médias das suites geradas usando as ferramentas Randoop e Evosuite estão dispostas na Tabela 3.5 e Tabela 3.6, respectivamente.

Quando comparamos tais coberturas com as do primeiro experimento (Tabelas 3.1 e 3.2), notamos que houve um aumento das médias das coberturas para ambas ferramentas (Randoop e Evosuite) para os sistemas: HealthCard, JAccounting e JMock, o que decorreu das novas funcionalidades. Apenas houveram decréscimos nas coberturas da ferramenta Ran-

doop para o sistema Jmock. A fim de entender tal número, realizamos uma análise da estrutura do código, e verificamos que quase todos os métodos criam internamente ou recebem como parâmetro de entrada e/ou saída, objetos complexos. Além disso, existem métodos cuja visibilidade não é pública.

As suites de teste geradas estão disponíveis no nosso site [8]. Na seção seguinte (3.2.1), apresentaremos os resultados do presente estudo e realizaremos uma comparação com os resultados do primeiro 3.

Tabela 3.5: Média da coberturas das novas replicações Randoop por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
HealthCard	85,8%	76,9%
JAccounting	46,7%	27,3%
JMock	46,1%	23,2%

Tabela 3.6: Média da coberturas das replicações Evosuite por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
HealthCard	97,7%	96%
JAccounting	34,9%	32,4%
JMock	74,6%	74,2%

3.2.1 Resultados e Discussão

A Figura 3.10 apresenta os resultados do segundo estudo, o qual engloba os resultados para os três sistemas e, refere-se à taxa de detecção das faltas injetadas. Além disso, as versões mais recentes de ambas as ferramentas (Randoop e EvoSuite) foram utilizadas. Das 120

[8] https://sites.google.com/copin.ufcg.edu.br/automaticgeneratetests/

faltas injetadas, 75,8% foram detectadas pelas suites geradas por ao menos uma das ferramentas. Como podemos ver, mesmo com recursos adicionados as ferramentas, a taxa de detecção de faltas de refatoramento obteve melhoria de 7,5% (de 68,3% para 75,8%). Um teste de proporção mostra, com 95% de confiança, que apesar de haver uma melhoria numérica, não foi encontrada diferença significativa entre os resultados de ambos os estudos (p-value = 0,2497).

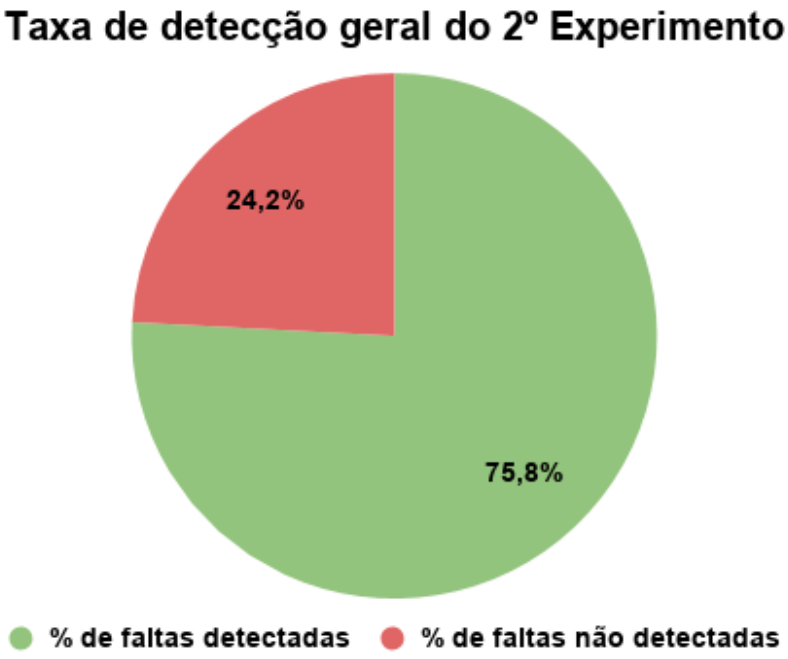

Figura 3.10: Proporção de faltas detectadas.

Dessa forma, estatisticamente, as novas versões de ferramentas são equivalentes quando se busca detecção de faltas de refatoramento. Para o presente estudo, consideramos a taxa de faltas não detectadas ainda bastante alta (24,2%), uma vez que faltas de refatoramento não detectadas podem dar ao desenvolvedor a falsa sensação de que edições seguras foram realizadas. Quando analisamos os resultados por ferramenta (Figura 3.11-a), podemos ver uma melhoria para a ferramenta EvoSuite. Onde, no primeiro estudo, esta detectou 64,2% das faltas injetadas, e sua nova versão detectou 73,3%. Essa melhoria é devido aos recursos adicionados recentes que tentam lidar melhor com objetos dependentes. Por outro lado, encontramos taxas piores para Randoop (uma queda de 13,3%). Esta queda é devido ao fato de que sua nova versão descarta *flaky test* e continua limitada quanto aos atributos complexos

e tipo de visibilidade não pública. Da mesma forma, quando analisamos as taxas de detecção agrupadas por tipo de falta (Figura 6-b), vimos que, para a falta do tipo *Troca Condição por Valor Fixo* houve um melhoria de 6,6%, enquanto que para a falta *Troca Operador da Condição* houve uma melhoria de 8,3%, na taxa de detecção.

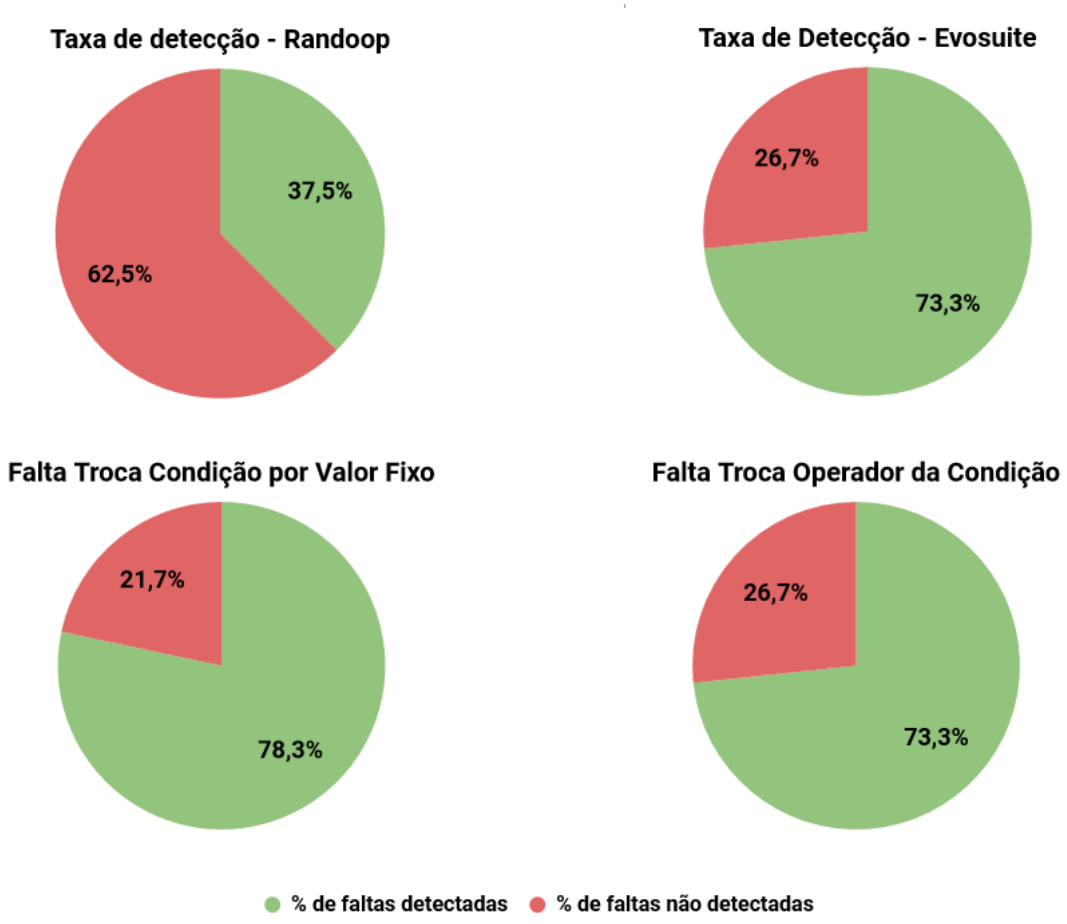

Figura 3.11: Resultado do segundo experimento por ferramenta (a) e por falta (b). Fonte: Autoria própria.

Devido ao grande número de faltas não detectadas, podemos responder Q1, afirmando que suites geradas somente não são completamente confiáveis para validar os refatoramentos, como também é necessário mais investigação. Aproximadamente 30% das faltas de refatoramento, não foram detectadas. Esta taxa permanece estável, mesmo para diferentes ferramentas e suas versões, além dos tipos de faltas. Isso pode levar os desenvolvedores a ter a falsa sensação de que eles realizaram edições de refatoramento seguros.

Casos de detecção específicos por ferramentas

Fazendo um levatamento das faltas que foram detectadas por apenas uma das ferramentas, temos que: dentre o casos detectados pelas suites da ferramenta Randoop e não pela Evosuite, houveram apenas 2 casos. Porém, quanto aos casos que foram detectados pelas suites da ferramenta Evosuite e não pela Randoop, obtivemos 50 casos, distribuídos nos dois tipos de faltas. Como exemplo de falta detectada apenas pelas suites Randoop, temos a falta que foi inserida no método *getAccount*, retirado do sistema *JAccounting*(Figura 3.12). Como exemplo das faltas detectadas pelas suites geradas pela ferramenta Evosuite, e não detectadas pela Randoop, temos a falta que foi inserida no método *byteArrayDataSource*(Figura 3.13). É válido ressaltar que o tipo de falta referente à ambos os casos, foi *Troca Condição por Valor Fixo*. Além disso, ambos exemplos representam os poucos casos onde as ferramentas conseguiram gerar casos de testes para objetos, de entrada e saída, não primitivos. Ao verificar tais casos, onde as faltas foram detectadas por apenas uma das ferramentas, optamos por propor uma solução que atenda à combinação de ambas ferramentas, ou seja, para um contexto de uso combinado da ferramenta Randoop e da Evosuite.

Código Fonte 3.8: Método *getAccount*, cuja falta foi inserida e detectada pelas suites Randoop.

```
1 public static Account getAccount(SQLFactory factory, Connection conn, int
      id) throws SQLException {
2         Account ret = null;
3         if (id == 0) {
4                 ...
5         } else {
6                 ...
7         } ...
8 }
```

Figura 3.12: Método cuja falta foi detectada apenas pela ferramenta Randoop.

Código Fonte 3.9: Método *byteArrayDataS*, cuja falta foi inserida e detectada pelas suites Evosuite.

```
1 private void byteArrayDataSource(InputStream aIs, String type) throws
    IOException{
2         ...
3         if (Bis != null){
4                 ...
5         }
6         ...
7 }
```

Figura 3.13: Método cuja falta foi detectada apenas pela ferramenta Evosuite.

3.3 Nova Unidade Experimental

A fim de expandir nossos resultados, acrescentamos, ao experimento, uma nova unidade experimental, totalizando quatro sistemas. Com relação a esta unidade experimental, trata-se do sistema *Transacted Memory* ($\approx$ 2KLOC) [30], uma funcionalidade específica para *API Javacard*. Em que esta é uma tecnologia que fornece um ambiente seguro para aplicativos que são executados em *smart cards* e outros dispositivos confiáveis com memória limitada e recursos de processamento [9]. Toda a configuração utilizada no primeiro estudo 3.1.1, também foi mantida. As coberturas médias das suites geradas usando as ferramentas Randoop e Evosuite estão dispostas na Tabela 3.7 e Tabela 3.8, respectivamente.

Quanto as médias das coberturas do sistema *Transacted Memory*, podemos observar que, tanto para a ferramenta Randoop quanto para a Evosuite, a média é considerada muito baixa. No geral, menos de 30% do código foi coberto. Ao analisarmos o código, notamos que os fatores encontrados nos dois estudos descritos anteriormente, persistiram para o novo sistema.

3.3.1 Resultados e Discussão

A Figura 3.14 apresenta os resultados obtidos do sistema *Transacted Memory*, adicionalmente aos resultados do segundo estudo. Engloba os resultados dos quatro sistemas e,

[9] http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html

Tabela 3.7: Média da coberturas das novas replicações Randoop por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
Transacted Memory	17,2%	7,1%

Tabela 3.8: Média da coberturas das replicações Evosuite por unidade experimental.

Sistema	**Cobertura de Linhas**	**Cobertura de Caminhos**
Transacted Memory	38%	29,4%

refere-se à taxa de detecção das faltas injetadas. Quando comparamos o resultado do primeiro estudo com o segundo, já adicionando a nova unidade experiemntal, temos que: das 160 faltas injetadas, 69,8% foram detectadas pelas suites geradas por ao menos uma das ferramentas. Como podemos ver, mesmo com recursos adicionados as ferramentas, a taxa de detecção de faltas de refatoramento obteve pequena melhoria de 1,1% (de 68,3% para 69,4%).

Um teste de proporção mostrou, com 95% de confiança, que apesar de haver uma melhoria numérica, não foi encontrada diferença significativa entre os resultados de ambos os estudos (p-value = 0,9554), isso considerando os resultados do primeiro estudo e do segundo, quando adicionado a nova unidade experiemental. Com isso, podemos reafirmar que as limitações encontradas nas ferramentas, referente a atributos complexos e tipo de visibilidade não pública, permaneceram nas novas versões das ferramentas. Mesmo quando expandimos o estudo para uma nova unidade experimental.

Ameaças à Validade

As ameaças apresentadas na Seção 3.1.9, aplicam-se ao presente estudo exploratório.

Figura 3.14: Resultado do segundo experimento por ferramenta (a) e por falta (b). Fonte: Autoria própria.

3.3.2 Considerações Finais

No presente capítulo apresentamos dois estudos exploratórios, ambos com a mesma configuração e realizados com o objetivo de investigar o uso de duas das principais ferramentas de geração de testes de unidade: Randoop e Evosuite, na detecção de faltas de refatoramento, bem como suas limitações. Nós utilizamos dois tipos de faltas, tais como: tipo *Troca Condição por Valor Fixo* e *Troca Operador da Condição*. No primeiro estudo, 68,3% das 120 faltas injetadas, foram detectadas pelas suites. Os demais 31,7% de faltas não detectadas, provam que tais ferramentas são limitadas. Tais limitações correspondem a criação de casos de testes para métodos de visibilidade não pública e/ou que recebam, como parâmetro de entrada ou saída, atributos de tipos não primitivos. Quanto ao segundo experimento, tratasse da replicação do primeiro, utilizando as versões mais novas das ferramentas. Verificamos que, mesmo com as versões mais atualizadas, apenas 69,4% das 160 faltas foram detectadas faltas foram detectadas. O que acabamos por confirmar, a partir de um testes de proporção, que não houve diferença estatística para o propósito do nosso trabalho, entre as versões das ferramentas. Com isso, podemos concluir que as ferramentas Randoop e Evo-

suite permanecem limitadas quanto aos fatores mencionados anteriormente. Isso pode levar o desenvolvedor a falsa sensação de ter realizado um refatoramento corretamente, enquanto que a suites de testes geradas automaticamente utilizadas como forma de validar o que está sendo desenvolvido, pode não ser suficeentemente eficaz na detecção, havendo a necessidade de outro meio auxiliar, que aumente a confiabilidade de realização de um refatoramento.

Capítulo 4

Prevendo a efetividade das ferramentas de Geração

Para entender quais fatores podem influenciar os processos de geração realizados pelas ferramentas nos estudos apresentados no capítulo anterior, investigamos manualmente cada caso em que a suite gerada foi e não foi capaz de detectar as faltas injetadas. Esta análise nos ajudou a identificar que, em geral, as faltas detectadas estavam localizadas em métodos com visibilidade pública e a maioria das faltas não detectadas estavam em métodos não públicos (*private* ou *protected*). É válido ressaltar que as faltas em métodos não públicos podem ser indiretamente detectadas, já que esses métodos podem ser chamados por outros métodos públicos e estes são cobertos pelos testes gerados. Isso pode acontecer, no entanto, em nosso estudo descobrimos que é pouco provável de acontecer.

Outro fator encontrado foi que as ferramentas não lidam bem com objetos complexos passados como parâmetros e/ou retornados em um método. Por objeto complexo, queremos dizer objetos de tipos não primitivos, que não são *java.lang.Object*. Durante o processo de geração, as ferramentas tentam criar objetos válidos usando um conjunto de dados pré-definido. Quando, por qualquer razão isso, não pode ser feito (*e.g.*, o objeto criado não alcançou um determinado estado necessário), o processo é abortado e outro método é escolhido para testes. Por exemplo, Figura 4.1 mostra o método `dupeCharge`, do sistema JAccounting. O qual requer um valor útil do tipo `Session` para ser executado, que não conseguiu ser gerado automaticamente por nenhuma das ferramentas. Assim, esse método permaneceu sem teste.

Código Fonte 4.1: Método *dupeCharge*, cujas ferramentas não criaram casos de teste

```
1 public static Charge dupeCharge(Session sess, Integer cid, Charge charge,
      Date newdate) throws ...{
2     Integer chargeId = charge.getId();
3     charge = dupeChargeNonRecurrence(sess, cid, charge, newdate);
4     Recurrence rec = AccountingPersistenceManager.getRecurrence(sess,
          AccountingPersistenceManager.APP_ID_CHARGE+"", chargeId.toString())
          ;
5     if(rec != null){
6         rec.setItemKey("" + charge.getId());}
7     return charge;
8 }
```

Figura 4.1: Método não executado pelos testes gerados.

Em resumo, nossa análise encontrou três fatores-chave que podem influenciar a eficácia das ferramentas de geração automática, quando pretendemos usar suas suítes para detectar faltas relacionadas ao *Extract Method*:

- *Visibilidade do Método* (VM)
- *Tipo de Parâmetro* (TP)
- *Tipo de Retorno* (TR)

Na próxima seção, investigamos a validade desses fatores em um terceiro estudo empírico, e propomos modelos estatísticos para prever a confiança da suíte gerada pelas ferramentas.

4.1 Modelos de Regressão

Sabendo-se que as ferramentas de geração de testes, Randoop e Evosuite, possuem limitações, considere o cenário onde: suponha que João é um desenvolvedor e aplicou um refatoramento do tipo *extract method* no código e, com isto, precisa validar se não introduziu no código alguma alteração semântica indesejada. João poderia confiar, seguramente, nas suites geradas automaticamente por tais ferramentas? Combinando desde a geração e

execução, as suítes automatizadas podem levar muito tempo (*e.g.*, em nosso estudo, todo processo para obtenção das suites de teste, levou uma média de 115 minutos). Dessa forma, seria útil se houvesse uma maneira de decidir ajudar João a decidir, antes mesmo antes de utilizar de fato as ferramentas, se os conjuntos de testes gerados seriam confiáveis nesse contexto. Podemos afirmar tal problema como segue:

> *Sendo P um programa; MC um conjunto de características do método refatorado de P; TS uma suite de testes gerada para P usando Randoop ou EvoSuite; PD a probabilidade de TS detectar uma falta de refatoramento. Nosso* problema *é encontrar $f : (TS, MC) \rightarrow PD$, uma função que decide, sem executar o conjunto, se o conjunto gerado seria útil para detectar qualquer falta de refatoramento recentemente introduzida.*

Com o objetivo de lidar com esse problema, derivamos um conjunto de modelos de predição baseados nos fatores-chave encontrados como importantes para determinar a eficácia das suites geradas automaticamente por ferramentas: visibilidade do método (VM), tipo de parâmetros (TP), e tipo de retorno (TR). Esses modelos de regressão levam em consideração os dados do nosso estudo e fornecem, de acordo com a característica do local onde o *extract method* foi realizado, uma predição se as ferramentas de geração (Randoop e EvoSuite) gerariam conjuntos de testes que são, ou não, prováveis de detectar uma falta de refatoramento do tipo *extract method*, se houver.

Em um cenário real, um desenvolvedor pode ter que decidir se quer usar o Randoop, o EvoSuite ou os dois [1]. Portanto, derivamos três modelos de predição, para cada caso: o *modelo Randoop*, o *modelo EvoSuite* e o *modelo combinado*. Para isso, aplicamos a Regressão Logística [21; 6]. A Regressão Logística é um método estatístico usado para prever uma resposta binária usando pelo menos um preditor categórico. No nosso caso, uma observação é uma suite de teste, os preditores (variáveis independentes) referem-se aos fatores-chave (1 se o método do refatorado tiver o fator, 0 caso contrário). É válido mencionar que classificamos como 1, os fatores-chave de: visibilidade do método ser pública, tipos de parâmetros e o tipo de retorno, serem não primitivos. Caso contrário (visibilidade não pública, parâmentros e retorno primitivos), atribuímos 0 para cada um destes.

[1] Os resultados do nosso estudo mostraram que algumas faltas foram detectadas apenas por suites Randoop, enquanto outras por suites EvoSuite. Assim, uma suite combinada pode detectar um conjunto maior de faltas.

Para a construção dos modelos, utilizamos amostras aleatórias do conjunto de dados dos nossos resultados do segundo estudo, o experimento de replicação. Começamos com um conjunto de treinamento de 10%, como de costume em tal análise, e aumentamos o tamanho da amostra até que o modelo ficasse estável, ou seja, quando seu valor *R-squared* atinge seu valor máximo. Para isso, usamos o ambiente R para computação estatística.

Tabela 4.1 mostra os coeficientes de cada preditor no caso do Modelo Combinado e seus *p-values*, enquanto a Equação 4.1 apresenta o modelo preditor proposto. Este modelo pretende inferir se uma suite gerada pelo Randoop ou EvoSuite é provável de detectar uma falta de refatorameno do tipo *extract method*, quando ela existir. Quanto mais próximo de 1 for o D, maiores são as chances de uma falta de refatoramento ser detectada. Na coluna p-value da Tabela 4.1, podemos ver que todos os fatores listados são de fato significativos (com 90% de confiança), e todos eles têm efeito sobre o resultado do sucesso. Da mesma forma, derivamos também o modelo de predição para ambos o Randoop (Tabela 4.2 e Equação 4.2) e EvoSuite (Tabela 4.3 e Equação 4.3).

$$D = 1/(1 + e^{-(1.1002+0.3203*VM-0.4146*TP-0.2090*TR)}) \tag{4.1}$$

$$D = 1/(1 + e^{-(0{,}2430+0.2833*VM-0.1888*TP-0.3383*TR)}) \tag{4.2}$$

$$D = 1/(1 + e^{-(0.6511-0.2411*VM-0.2324*TP-0.2644*TR)}) \tag{4.3}$$

Tabela 4.1: Regressão estatísticas para o modelo combinado

	Coefficient	**p-value**
Intercept	1.1002	6.31e-09
VM	0.3203	0.059354
TP	0.4146	0.000184
TR	0.2090	0.072746

Tabela 4.2: Regressão estatísticas para o modelo do Randoop

	Coefficient	**p-value**
Intercept	0.2430	0.14613
VM	0.2833	0.09578
TP	0.1888	0.06305
TR	0.3383	0.00212

Tabela 4.3: Regressão estatísticas para o modelo EvoSuite

	Coefficient	**p-value**
Intercept	0.6511	9.97e-06
VM	0.2411	0.0985
TP	0.2324	0.0230
TR	0.2644	0.0259

4.1.1 Avaliando os Modelos

Usando conjunto de teste

Para verificar o ajuste de nossos modelos, primeiro os testamos usando os dados restantes do nosso conjunto de dados (sem sobreposição com o conjunto de treinamento). Com 95% de confiança, os modelos predisseram corretamente quando as suites geradas detectariam, ou não detectariam, as faltas de refatoramento com precisão de 52% considerando o modelo Randoop, 77% para o modelo EvoSuite e 75% para o modelo combinado .

Usando Novo Estudo Empírico

Em uma segunda análise, investigamos o desempenho de nossos modelos de previsão em diferentes sistemas e conjunto de faltas. Assim, realizamos um terceiro estudo empírico que replicou o estudo descrito no Capítulo 3, porém com as versões mais recentes das ferramentas Randoop e Evosuite. Os novos sistemas *open-source* utilizados, foram:

- Mondex ($\approx$ 1KLOC)[2], sistema que emula uma bolsa eletrônica hospedada em um *smart card*;

[2]http://vsr.sourceforge.net/mondex.htm

- Samples ($\approx$ 4KLOC)[3], um conjunto de programas para ajudar os desenvolvedores a aprender como especificar restrições;

- OpenJMLDemo ($\approx$ 10KLOC)[4], uma ferramenta projetada para verificar restrições em programas Java.

Para esses sistemas, 120 faltas, de cada tipo, foram injetadas seguindo os procedimentos descritos no Capítulo 3. Então, nós geramos suites de regressão geradas automaticamente usando Randoop e Evosuite, e verificamos se os modelos de predição, de fato, prediziam a capacidade de detecção de faltas, das suites. Para isso, estabelecemos a seguinte regra de limiar: valores de D superiores a 0,5 indicam que um conjunto gerado *poderia* detectar faltas no método refatorado; Caso contrário, a suite passaria. Sabendo-se que D varia de 0 até 100, nós definimos o nosso ponto médio de 0,5 em que os valores acima de 50% tendem fortemente a detectar faltas de refatoramento em um dado método.

Enquanto o modelo Randoop previu corretamente apenas $\approx$ 52% dos casos, o modelo EvoSuite teve uma taxa de sucesso de 84% e o combinado $\approx$ 86% (Figura 4.2). Esses resultados mostram que os fatores-chave são de fato influenciadores para a capacidade das ferramentas de gerar conjuntos adequados para detectar faltas de refatoramento (RQ2). No entanto, eles parecem ser mais importantes para os resultados do EvoSuite. Porém, apesar dessa diferença com relação ao modelo Randoop, isso não exclui tal abordagem. Visto que, a escolha de uso de cada modelo não está relacionada unicamente a taxa de acerto de predição, mas também ao contexto e abordagem ao qual o desenvolvedor está utilizando ou pretende utilizar.

Investigamos as taxas mais baixas alcançadas pelos modelos. Nossa análise mostrou que o modelo Randoop bastante bem com relação aos Verdadeiros Negativos (previu que uma suíte não detectaria uma falta e realmente não detectou), 62 casos previu corretamente, mas criou um grande número de Falsos Negativos (previu que uma suite não detectaria uma falta, mas, de fato, houve), 58 casos. Com relação aos Falsos Negativos, podemos observar que, para o modelo Evosuite, houveram 19 casos. Quando ao modelo Combinado, apenas 17 casos foram previstos erroneamente.

[3]https://jaccounting.dev.java.net/

[4]https://github.com/OpenJML/OpenJMLDemo

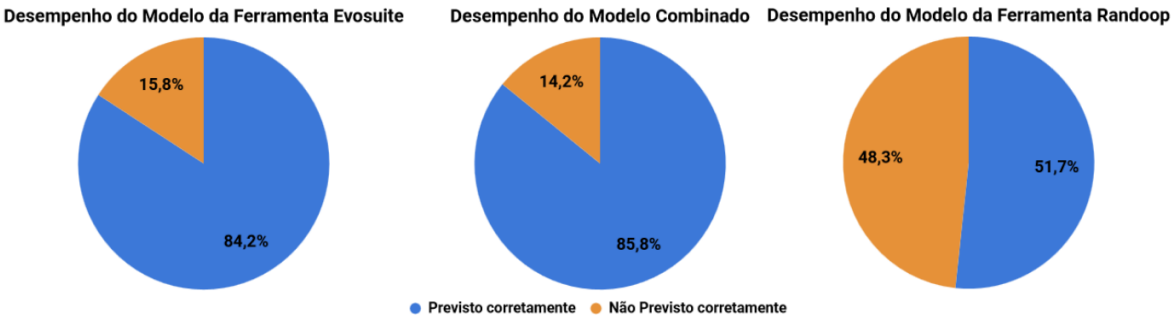

Figura 4.2: Desempenho do Modelos Propostos

Uma vez que os fatores usados no modelo são estatisticamente significativos (Tabelas 4.1 - 4.3), acreditamos que, para Randoop, esses modelos precisam ser refinados usando técnicas (*e.g.*, aprendizagem de máquina) ou precisa ser complementado considerando um conjunto maior de fatores.

4.1.2 Diretrizes de Uso dos Modelos de Predição

Nesta seção, listaremos algumas diretrizes de como usar os modelos descritos na Seção 4.1. As ferramentas de geração automática de teste, podem de fato ser usadas para validar edições de refatoramento [26; 36]. No entanto, os desenvolvedores devem estar cientes de suas limitações práticas. Com isso, apresentamos um conjunto de fatores relacionados às limitações das ferramentas e três modelos de predição, que podem ser usados para decidir se a suite gerada, é segura e confiável. Portanto, sugerimos que estes sejam usados na prática da seguinte maneira: suponha que John seja um desenvolvedor e que tenha aplicado recentemente um *extract method* em seu projeto. Agora, ele precisa validar se introduziu, ou não, faltas no código anteriormente estável. Como o projeto de John não tem uma suite de regressão, ele quer saber se pode usar suites de testes geradas para esse contexto, ou se ele deve trabalhar para criar manualmente, do zero, uma suite de regressão. Dessa forma, John, ao fornecer a informação de onde o refatoramento foi realizado (a visibilidade do método, o tipo dos parâmetros e o tipo do retorno), pode primeiro usar o Modelo Combinado (Equação 4.1). Se ele obtiver valores de D maiores que 0,5, isso indica que a suite gerada provavelmente detectaria faltas de *Extract Method*, se houverem em, relacionada ao refatoramento aplicado por John. Caso contrário, os resultados das suites geradas podem ser enganosos e ele deve direcionar esforços à uma estratégia de validação diferente, como uma suite de

testes manuais. Sendo as suites geradas uma opção válida (D $>$ 0,5), John pode decidir melhor qual ferramenta usar, uma vez que cada ferramenta possui diferentes recursos e tempos de resposta. Para isso, ele deve fazer uso do modelo de predição da respectiva ferramenta (modelo Randoop 4.2 e/ou modelo EvoSuite 4.3).

Capítulo 5

REFANALYZER

REFANALYZER é uma ferramenta que objetiva ajudar os desenvolvedores a decidir quando confiar em suites geradas para validar refatoramento. Para tal, sua utilização pode ocorrer de duas formas, que variam de acordo com as entradas fornecidas. A Figura 5.1 apresenta uma visão geral da REFANALYZER.

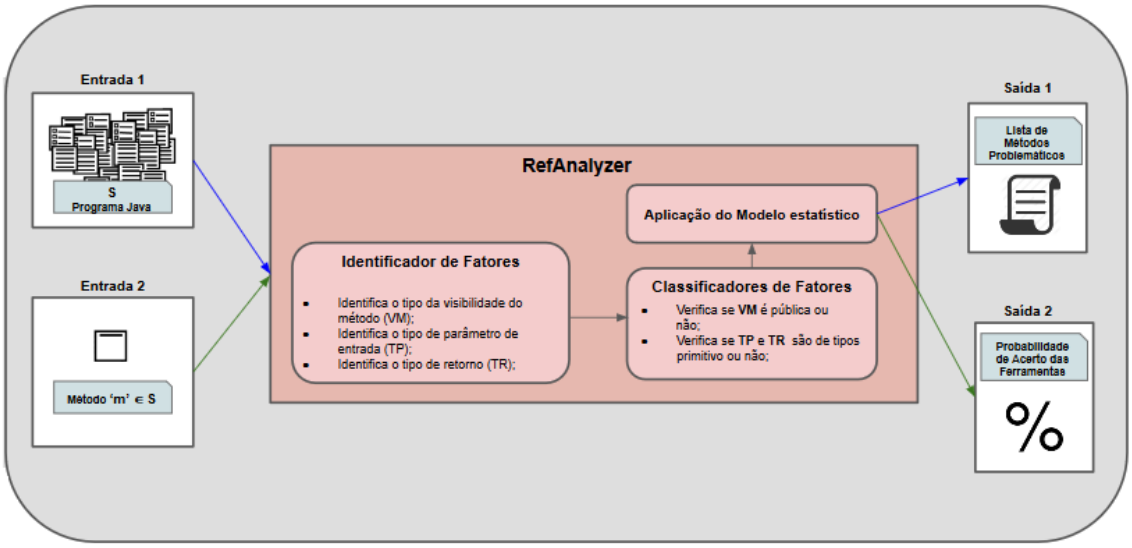

Figura 5.1: Visão geral do procedimento realizado pela ferramenta REFANALYZER. Fonte: Autoria Própria.

Na primeira forma de uso, REFANALYZER receber um projeto JAVA e fornece como saída, uma lista de métodos classificados como problemáticos. Um método é denominado como problemático, quando a capacidade das ferramenta gerarem casos de testes para este, que detecte uma falta de refatoramento, for menor que 50%. Na segunda forma de

uso, o desenvolvedor fornece a informação de onde o refatoramento foi realizado, a partir dessa informação, a REFANALYZER informa a probabilidade das ferramentas de geração automática de testes, gerarem casos que detecte uma falta, no referido local.

Quanto ao funcionamento da REFANALYZER, está divido em três partes distintas e aplica-se para formas de uso. A primeira parte é o módulo **Identificação de Fatores**, nesta etapa o código JAVA lido e interpretado, de modo que para cada método os fatores correspondentes a visibilidade do método e tipos de parâmetro de entrada e de retorno, são extraídos. Para isto, utilizamos a API de java *Reflection*, que é comumente utilizada em sistemas que requerem capacidade de examinar ou modificar o comportamento do tempo de execução dos aplicativos em execução na máquina virtual Java [1].

Em seguida, está informação é passada para o módulo de **Classificadores de Fatores**. Neste, vericamos a visibilidade, e os tipos dos parâmetros de entrada e de retorno. Dessa forma, nós utilizamos nomenclaturas padronizadas para tais fatores, onde **VM** refere-se a visibilidade do método ser pública ou não, **TP** refere-se ao tipo de parâmetro recebido como entrada ser do tipo complexo ou não primitivo e, por fim, **TR** refere-se ao tipo do retorno ser não primitivo.

De posse desta informação, temos o terceiro e último módulo, **Aplicação do Modelo Estatístico**. Onde, para cada método identificado como problemático, é calculada a probabilidade das ferramentas gerarem casos de testes que detectem uma falta de refatoramento do tipo *extract method*. Como três modelos de predição foram criados (Seção 4), também geramos três versões para REFANALYZER, a que implementa o modelo de predição da Análise Combinada (eq. 4.1), do Randoop (eq. 4.2) e do Evosuite (eq. 4.3).

5.1 Avaliação Preliminar da Ferramenta

Os estudos exploratórios descritos nas seções anteriores, se concentraram na identificação das limitações das ferramentas Randoop e Evosuite. A partir deles, propusemos um conjunto de modelos estatísticos de predição da probabilidade das ferramentas gerarem casos de testes que detectem uma falta de refatoramento e, por fim, o último estudo consistiu na validação destes modelos. No entanto, surgiu a necessidade de avaliar REFANALYZER, de modo à

[1] https://docs.oracle.com/javase/tutorial/reflect/

saber o quão útil nossas proposições podem ser na validação de refatoramento dentro do contexto de projetos ágeis.

Dado que, na metodologia ágil, o tempo entre iterações é curto e suites criadas manualmente nem sempre estão disponíveis. Então, uma das principais alternativas utilizadas, são suites de regressão geradas automaticamente por ferramentas. Com isso, surguiram as seguintes questões: Quando confiar em suites de geradas? O que fazer para aumentar a eficiência das ferramentas de geração? REFANALYZER busca atuar nessas duas frentes.

Para avaliar os benefícios práticos do uso de REFANALYZER, realizamos um estudo com desenvolvedores profissionais. Este estudo tem como objetivo avaliar a capacidade da ferramenta em ajudar os desenvolvedores a decidir quando usar suites geradas. A seção estará organizada da seguinte forma: (i) Perfil dos Participantes, (ii) Procedimento do Estudo, (iii) Tarefas de Inspeção e (iv) Análise e Discussão.

5.1.1 Perfil dos Participantes

No nosso estudo, recrutamos 30 desenvolvedores, sendo 29 do sexo masculino e 1 do sexo feminino, distribuídos em 8 projetos de *software* distintos de grande e médio porte. Antes da realização do estudo, os participantes preencheram o questionário (Apêndice A) referente ao seu *background*. Tivemos profissionais com diferentes níveis de experiência (Júnior, Pleno e Senior) e que desempenham diferentes funções dentro da equipe em que trabalham, tais como: Desenvolvedor, Técnico, Analista de Sistemas, (*Team Leaders*), Analista de Qualidade, Engenheiro de *Software*, Gerente de Projeto, Líder Técnico de Desenvolvimento. A Figura 5.2 sumariza as funções dos profissionais que participaram do estudo, na respectiva ordem.

A Figura 5.3, apresenta detalhes sobre o *background* dos participantes. Mais de 70% dos profissionais possuem mais de três anos de experiência com programação. Com relação a realização de edições de refatoramento, 45% dos participantes disseram que realizam diariamente ou pelo menos, uma vez na semana. Além disso, $\approx 70\%$ disseram que realizam refatoramento de forma completamente manual, $\approx 80\%$ revisam tais edição de forma manual e $\approx 70\%$ afirmaram que o tipo de teste mais utilizado nos projetos onde trabalham, é teste criado manualmente.

Fornecemos aos participantes uma lista com 16 tipos de refatoramentos e perguntamos

Figura 5.2: Visão geral do profissionais que participaram do estudo.

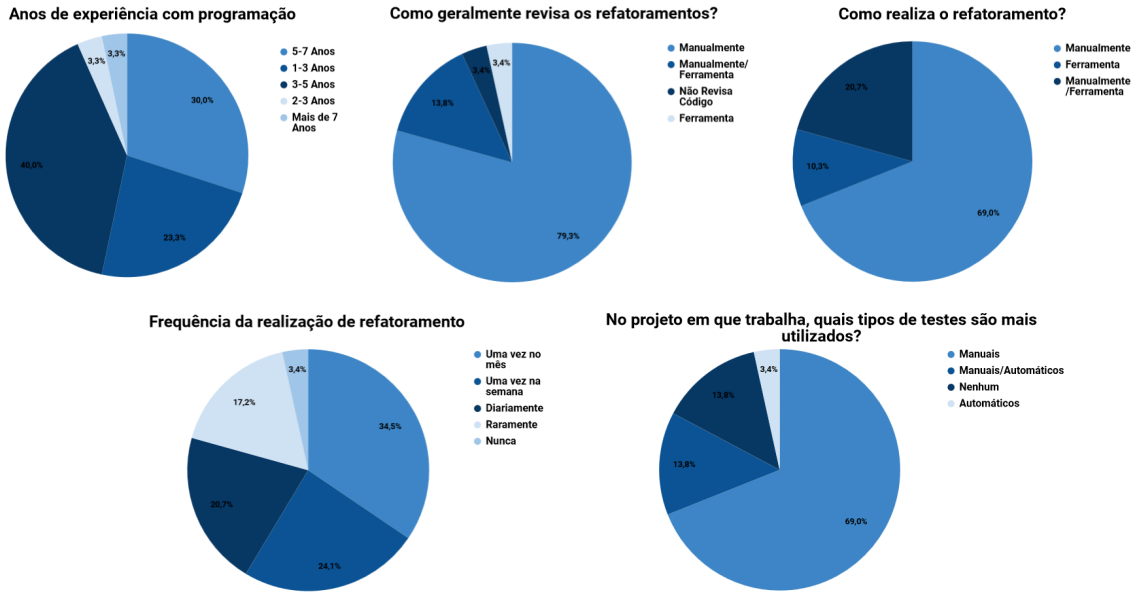

Figura 5.3: Visão geral do perfil dos profissionais.

quais, dentre os refatoramentos, lhes eram familiares. Como mostra a Figura 5.4, dentre o 16 tipos distintos, os mais comuns entre os profissionais foram: *Extract method*, *Rename Method*, *Extract Class*, *Move Method*, *Add Parameter* e *Encapsulate Field.*

É válido ressaltar que dos 30 profissionais que participaram do estudo, 90% possuem familiaridade com o refatoramento utilizado no nosso contexto, que é o tipo *Extract method*, e os outros 10% alegaram que realizam edições, porém não sabiam qual a nomenclatura atribuida por Fowler para cada um deles.

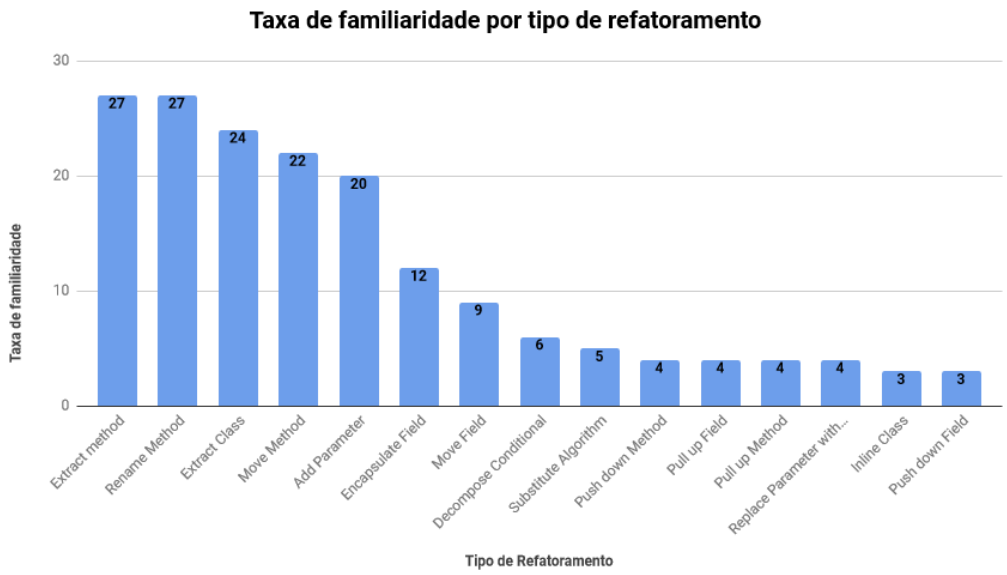

Figura 5.4: Visão geral da taxa de familiaridade do tipo de refatoramento

5.1.2 Procedimento do Estudo

Antes da iniciar a sessão do estudo, foi pedido aos participantes que respondessem o pré-estudo (Apêndice A), no qual o objetivo foi coletar os dados correspondentes ao perfil e *background* de cada participante (Figura 5.3). Depois disto, realizamos durante aproximandamente quinze minutos, uma explanação geral do estudo e como este estava dividido. Além disso, explicamos o propósito da ferramenta REFANALYZER.

Em seguida, cada participante recebeu, de forma impressa, os cenários (Apêndice B), estando estes divididos em três partes. Os dois primeiros cenários têm o propósito singular de analisar se o resultado fornecido pelo modelo de predição, influenciaria na decisão de aplicar ou não um refatoramento do tipo *Extract Method*, se aumentaria a confiabilidade para aplicá-lo e qual alternativa o desenvolvedor tomaria para validá-lo. Vale salientar que, tal resultado consiste na probabilidade das suites de testes geradas automaticamente, detectarem uma falta no refatoramento. Diante disso, cada cenário apresenta valores de predição de níveis diferentes. O primeiro cenário 48%, número relativamente baixo e que indica que provavelmente as suites de teste geradas tem pouca chance de identificar uma falta no refatoramento, e o segundo, foi 72%, este número indica que as suites tendem a ser efetivas. No terceiro e último cenário, utilizamos REFANALYZER fornecendo como entrada um retorno produzido

pelo sistema JMock. Como resultado, a ferramenta apresenta uma lista de métodos, onde estes são classificados como problemáticos para a geração de testes automáticos. Diante disso, gostaríamos de saber se as saídas fornecidas seriam de fato úteis e quais medidas o desenvolvedor tomaria, com estas informações.

5.1.3 Questões de Pesquisa

Para avaliar a capacidade da REFANALYZER em ajudar os desenvolvedores a fazer um refatoramento de forma mais confiável, e para guiar nossa investigação, definimos as duas questões de pesquisa descritas abaixo.

- RQ1: As informações fornecidas pela REFANALYZER, influenciam na realização ou não de refatoramentos?
- RQ2: As informações fornecidas pela REFANALYZER, influencia na confiabilidade dos participantes com relação aos testes utilizados no projeto?

5.1.4 Análise e Discussão

A presente seção trata da análise e discussão das informações obtidas a partir da segunda parte do questionário (Apêndice B).

Cenários 1 e 2

Nos cenários 1 e 2, foram apresentados métodos que precisariam de melhorias estruturais, a partir de um *extract method*. Além disso, foi informado que um modelo de predição, previu que 48% e 72%, respectivamente, das suites de teste geradas automaticamente, por ferramentas de geração, identificarão uma falta de refatoramento, caso seja inserida durante a aplicação do refatoramento. Números estes relativamente diferentes, um considerado baixo, que indica que provavelmente as suites de teste geradas tem pouca chance de identificar uma falta no refatoramento, e outro considerado alto, que indica o contrário. Para avaliar a perspectiva dos desenvolvedores quanto à utilidade da REFANALYZER, pedimos que avaliassem entre uma das cinco opções: Inútil, Pouco Útil, Neutro, Útil ou Muito Útil.

Um visão geral do primeiro cenário, onde a probabilidade, que foi dada a partir do modelo, foi de 48%. Obtivemos $\approx 67\%$ de avaliação positiva, ou seja, tivemos 4 profissionais que avaliaram como Muito Útil, 16 como Útil. Em contrapartida, 3 avaliaram como Neutro, 6 como Pouco Útil, e 1 avaliou como Inútil. Além disso, grande parte afirmou que para o contexto de detecção de refatoramento, confiam parcialmente ou não confiam na efetividade das suites geradas. Apesar de 17 deles afirmarem que seguiriam com a aplicação do refatoramento, 9 afirmaram que postergariam o refatoramento para um outro momento e 4 desistiriam do refatoramento. Por fim, $\approx 80\%$ disseram que para este caso de validação, fariam uso de suite gerada, porém combinada com uma suite de testes criados manualmente.

Note que, apesar da taxa de avaliação não ser relativamente alta, $\approx 47\%$ dos profissionais adiariam ou cancelariam a aplicação do refatoramento, modificando assim a intenção de aplicar o refatoramento. Quanto a validação do segundo cenário, onde a probabilidade que foi dada a partir do modelo, foi de 72%. Obtivemos $\approx 90\%$ de avaliação positiva. Sendo que, 13 avaliaram como muito Útil, 14 como Útil e 3 como neutro. Além disso, 72% afirmou que para o contexto de detecção de refatoramento, confiam parcialmente ou confiam plenamente na efetividade das suites geradas. 24 deles afirmarem que seguiriam com a aplicação do refatoramento, 3 afirmaram que postergaria o refatoramento para um outro momento e 3 desistiriam do refatoramento. Por fim, mais $\approx 85\%$ disseram que para este caso de validação, fariam uso de suite gerada, porém combinada com uma suite de testes criados manualmente ou, combinadas com outra estratégia. Note que, em ambos cenário, podemos ver a utilidade do modelo de predição, $\approx 79\%$ de avaliação positiva, bem como a estabilidade e a instabilidade na confiança do profissional para com uma suite de teste gerada, após os resultados do modelo.

Cenário 3

Para avaliar a perspectiva dos desenvolvedores quanto à utilidade da REFANALYZER, pedimos que os participantes a avaliassem segundo entre uma das cinco opções: Inútil, Pouco Útil, Neutro, Útil ou Muito Útil. A figura 5.5 apresenta a opinião dos 30 profissionais com relação a informação fornecida. 14 participantes avaliaram a taxa de confiabilidade das ferramentas de geração criadas pelo REFANALYZER como Muito Útil, 13 como Útil, 2 como Neutro, 1 como Pouco Útil e 0 como Inútil.

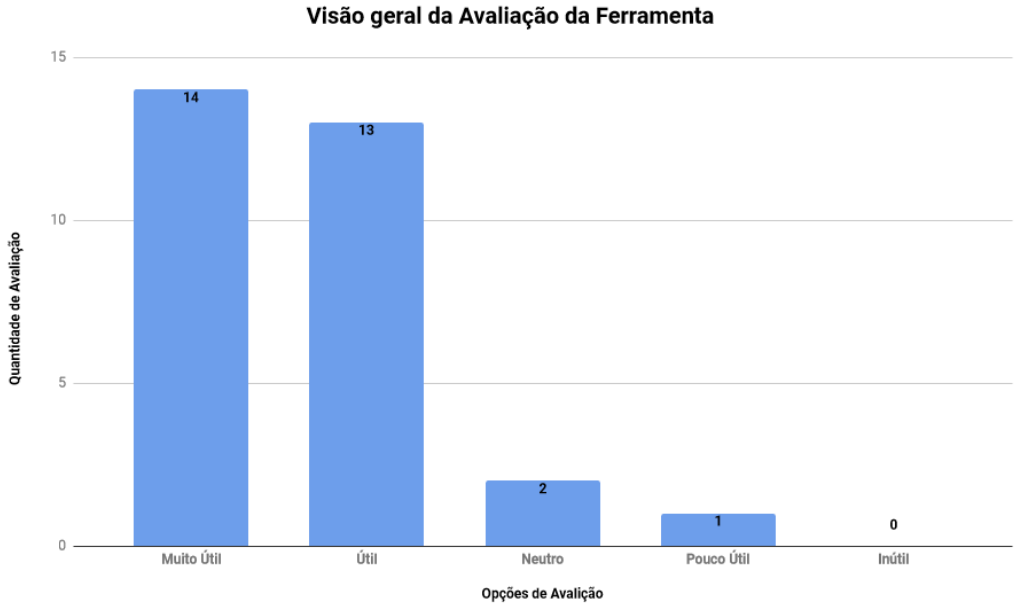

Figura 5.5: Avaliação da REFANALYZER

Note que não houve nenhuma avaliação como Inútil e houve apenas uma pessoa disse que a informação é pouco útil. Ao analizarmos esse único caso, vimos que o desenvolvedor não compreendeu o problema e, consequentemente, a relevância da informação. A justificativa apresentada por ele, foi:

"Se eu não conheço o total dos métodos existentes, essa informação tem pouca utilidade. Se houveram 15.000 métodos, está ótimo. Se houverem 40 métodos, está péssimo. Este informação teria que vir junto a uma porcentagem."

Em contrapartida à esta informação, quando a ferramenta apresenta uma lista de métodos problemáticos, isso quer dizer que todos os métodos que estão na lista, possuem probabilidade inferior a 50% dos testes gerados serem eficazes na detecção da falta de refatoramento. Diante disso, a quantidade de métodos não importa, visto que o propósito dos testes é assegurar que estabilidade do sistema, e do teste de regressão é verificar que nenhum *bug* foi inserido.

Quando paramos para analisar os dois casos onde os participantes avaliaram como Neutro, vimos que um não apresentou nenhuma justificativa e o segundo disse: *"Porque não saberia como sair, com esta informação"*. Ou seja, não sabe o que fazer com a informação. Novamente nos deparamos com limitação de entendimento ou conhecimento técnico por

parte do participante. Quando analisamos os casos onde os participantes avaliaram como sendo Muito Útil e Útil, 90% do total de avaliações, nos deparamos com as avaliações interessantes de profissionais que executam diferentes funções, tais como:

"Gerente de Projeto: Auxilia o desenvolvedor na tomada de decisão e do momento certo para ser feito um refatoramento no código. A fim de diminuir a complexidade dos métodos listados."

"Lider Técnico de Desenvolvimento: Importante para alertar o desenvolvedor que ele deve ter mais cuidado ao tratar cenários que contenham esses métodos."

"Analista de Sistema (Team Leader): Dessa forma, é possivel definir outra extratégia de teste para cada metodo descrito nesta ferramenta, melhorando assim a confiabilidade no teste após realizar o refatoramento."

"Desenvolvedor: Ela funciona como um norte para melhorar os testes."

"Técnico: Essa informação é útil para identificar métodos que possivelmente receberão mais análise e testes manuais do que automáticos."

"Analista de Sistema: Repensaria a possibilidade de realizar o refatoramento ou então fazê-lo escrevendo casos de testes manualmente, para os métodos que foram listados"

Assim, podemos responder RQ1 e RQ2 afirmando que, para o contexto do nosso estudo, os desenvolvedores acham que a REFANALYZER influencia na tomada de decisão em fazer ou não o refatoramento. Além disso, também influencia na confiabilidade com relação aos testes utilizados no projeto, para garantir a corretude do refatoramento, propondo a criação de testes manuais.

A figura 5.6 apresenta uma disposição geral da avaliação fornecida pelos profissionais pela frenquência com que estes aplicam um refatoramento. Nesta, podemos ver uma não padronização da avaliação quando relacionado ao tipo da função desempenhada, e esta variação se aplica também à frequência em que os participantes aplicam um refatoramento.

Note que participantes de todas as funções avaliaram Muito Útil e Útil, independentemente se a frequência com que realizam um refatoramento foi: Nunca, Diariamente, Uma vez na Semana, Uma vez no Mês ou Raramente. Ponto muito positivo para a REFANALYZER, pois independentemente dos profissionais das 8 categorias, ela obteve 90% de avaliação positiva. Com isso, validamos a REFANALYZER, obtendo indícios que esta auxilia de forma positiva no desenvolvimento do projeto, além de trazer mais confiança ao desenvolvedor em

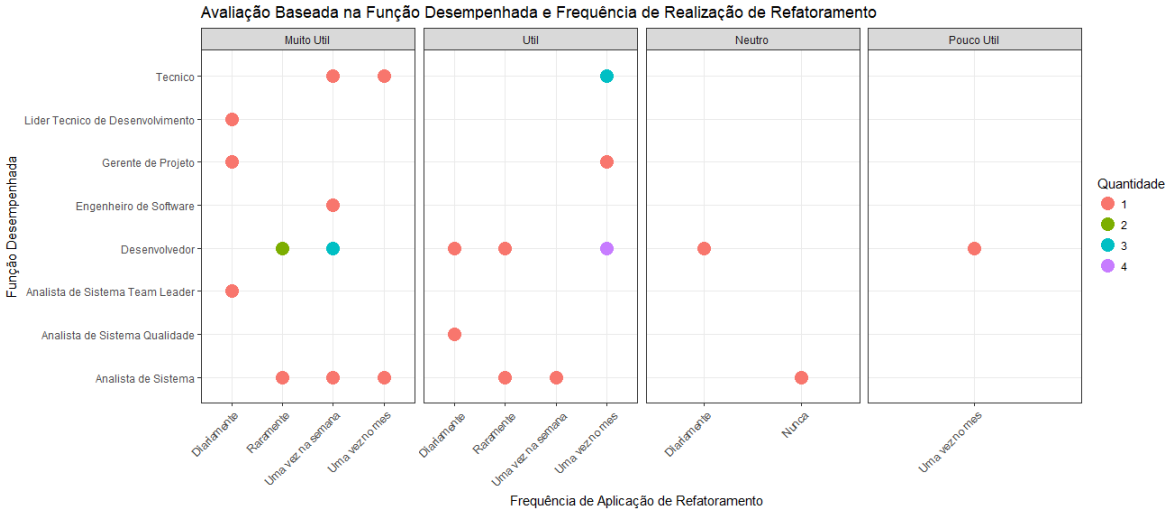

Figura 5.6: Avaliação da REFANALYZER baseada na função desempenhada

realizar um refatoramento do tipo *extract method*.

5.1.5 Limitações e Melhorias

A fim de melhorar o uso e aplicabilidade da REFANALYZER, nós elencamos algumas melhorias, que ajudam diretamente na minimização das limitações e ampliação da aplicabilidade da mesma. O primeiro ponto refere-se ao escopo de tipos de refatoramentos que a REFANALYZER se aplica. Como melhoria, pretentemos aplicar para outros tipos de refatoramentos. Uma melhoria que aborda as saídas fornecidas pela REFANALYZER, seria a criação de *stubs* de testes para cada método problemático, de modo a facilitar o desenvolvedor a cria o casos de teste manuais. Atualmente, REFANALYZER não possui nenhum tipo de interface gráfica, como melhoria podemos criar uma interface que viabilise e facilite o uso da mesma. Por fim, como melhoria futura, a REFANALYZER poderia se tornar um *plugin* para a IDE Eclipse. De tal modo a detectar automaticamente o local de onde o refatoramento foi inserido e fazer a análise em *background*. Além disso, pretendemos melhorar a REFANALYZER que implementa o modelo Randoop, a partir da utilização de outras estratégias estatíticas, para aumentar a capacidade de predição.

Capítulo 6

Trabalhos Relacionados

O presente capítulo trata-se dos trabalhos que se relacionam com o nosso. Além disso, o capítulo está dividido em três seções, tais quais: Ferramentas para Validação de Refatoramentos, Testes e Refatoramentos e Ferramentas de Geração de Testes.

6.0.1 Ferramentas para Validação de Refatoramentos

Edições de refatoramento geralmente requerem testes de regressão como forma de validar a preservação do comportamento externo do *software*, particularmente se o refatoramento for aplicado manualmente [27; 20]. Além disso, Daniel et al. [12] e Soares et al. [38] mostram que mesmo os refatoramentos aplicados com *refactoring engines* convencionais precisam ser validados, pois a corretude da implementação do refatoramento pode não ser garantida. Ambos usam testes nas suas técnicas para validar *refactoring engines*. Enquanto Daniel et al. usa testes como um dos oráculos para determinar se o comportamento é preservado ou não, Soares et al. propõe uma técnica baseada na geração automática de casos de teste usando a ferramenta Randoop. Além disso, Mongiovi et al. [26] estende o trabalho de Soares ao adicionar análise de impacto para orientar a geração de casos de teste. Embora suas técnicas tenham descoberto uma série de faltas em edições de refatoramento, realizadas por importantes ferramentas de refatoramento, o uso de suites de testes geradas automaticamente na prática, ainda merece maior investigação. Para o nosso trabalho, utilizamos testes de regressão para validar se o comportamento do *software* foi preservado, após a realização de um refatoramento.

6.0.2 Testes e Refatoramentos

A eficácia na validação de edições de refatoramento por suites de testes criadas manualmente, é investigada no estudo exploratório realizado por Alves et al. [3]. Eles procuram identificar deficiências em uma suite de testes para detectar faltas de edição em três projetos *open source*. Os resultados revelaram que as suites de testes criadas manualmente podem ser ineficientes para detectar faltas de refatoramento, exigindo o aumento da suite de testes, que pode ser realizado, por exemplo, via geração automática. Ainda dentro do contexto de validação de refatoramento, porém a partir de testes de regressão, onde estes servem como aparato de confiança às edições de refatoramento, suites de testes inadequadas podem prejudicar as decisões a serem tomadas pelo desenvolvedor. Com isso, Alves et. al. [5] propuseram a REFDISTILLER, uma ferramenta para detectar e localizar anomalias em um refatoramento inserido manualmente. Durante a avaliação, a REFDISTILLER identificou 97% de anomalias inseridas, dentre estas 24% não foram detectadas pelas suites de teste geradas. Por fim, os resultados mostraram que a REFDISTILLER pode ajudar a verificar a correção das refatoramentos manuais. A partir da análise feita por Alves et al., procuramos identificar as limitações, não para as suites manuais, mas para as suites de testes geradas automoticamente, similarmente, quanto a detecção de faltas de refatoramento.

Ferramentas de geração automáticas são avaliadas em termos de capacidade de revelação ou cobertura de faltas no geral. Mariano Ceccato et. al. [11] abordou a questão dos casos de teste gerados automaticamente, pelas ferramentas Randoop e Evosuite, serem igualmente eficazes no suporte à depuração como testes escritos manualmente. Enquanto que, em seu trabalho, Kim *et al.*[31] investigaram o impacto de edições de refatoramento no uso de testes de regressão, utilizando o hitórico de projetos Java. Para isto, avaliaram três projetos *open source* e viram que apenas 22% dos refatoramentos aplicados no estudo, são cobertos pelas suites de testes de regressão do programa. E 62% da suite de testes não exercita os refatoramentos aplicados. Diante dos nossos estudos exploratórios, mostramos que os casos de teste gerados automaticamente, por tais ferramentas, são tão úteis para depuração como os casos de teste manual.

6.0.3 Ferramentas de Geração de Testes

No contexto de teste de regressão, os testes são reexecutados com propósito de checar se nenhuma falta foi inserida, pois pode acarretar em uma modificação inesperada do sistema. Diante disso, Chang-ai Sun et. al [39] propuseram uma nova técnica para teste de regressão, baseada em uma família de estratégias de seleção de mutantes. Os resultados mostraram que a técnica proposta pode melhorar signitificamente a eficiência de diferentes atividades de teste de regressão, incluindo a redução de casos de testes e priorização.

Quanto à capacidade de detecção de faltas em geral, Kracht et al. [22] investigaram a qualidade de suites de teste geradas automaticamente quando comparadas às suites de testes manuais, com foco em 10 programas do *SF110 corpus12*. Os resultados mostram que o EvoSuite pode criar suites de testes de qualidade similar à escrita manual sobre cobertura de *branch* e *score* de mutação. Tal resultado, incentiva o uso da geração automática de casos de teste, uma vez que os custos de criação são menores. Além disso, Shamshiri et al. [34] realizaram um estudo empírico sobre a eficácia de suites de testes geradas automaticamente em detectar faltas. O estudo se concentra em três ferramentas de geração, incluindo EvoSuite e Randoop, e em faltas reais apresentadas na base de dados Defects4J. Os resultados mostram que as suites geradas detectaram 55,7% das faltas e apenas 19,9% da suite de teste detectaram uma falta. Devido ao fato de que as ferramentas têm algoritmos randomizados, uma falta pode não ser encontrada em todas as execuções das ferramentas. Na verdade, se a detecção requer que todas as execuções detectem a falta, as taxas de detecção são realmente baixas. A partir dos resultados, os autores sugerem melhorias que podem ser aplicadas às ferramentas em relação à criação de objetos complexos, otimização de cadeias, tratamento de condições complexas e métodos e atributos privados.

Além disso, Almasi et al. [1] apresenta os resultados de uma avaliação da geração de testes de unidade no âmbito de uma aplicação industrial. O estudo foca nas ferramentas EvoSuite e Randoop. Os resultados mostram que as suites de testes do EvoSuite detectaram até 56.40% das faltas, enquanto que as suites de testes Randoop detectaram 38%. Ao analisar os casos sem detecção, eles dependem de valores específicos ou de um estado complexo de objetos. Nós analisamos a capacidade das suites geradas automaticamente pelas ferramentas Randoop e Evosuite, em detectar falta de refatoramento. Os resultados mostraram que tais ferramentas são limitadas, quanto a lidar com parâmetros de entrada e saída do tipo não

primitivos, e com métodos de visibilidade não pública. Além disso, propomos algumas alternativas, a fim de aumentar a confiabilidade da aplicação de um refatoramento.

Apesar de apresentar desempenho equivalente quando comparado aos conjuntos de testes criados manualmente, pode-se notar que as suites de testes geradas automaticamente são limitadas com relação a capacidade de detecção de faltas. A limitação tem sido relacionada à baixa cobertura de elementos no código, como valores (complexos) e dependências de códigos e dados privados [40; 1]. Nesse sentido, Arcuri et al. [8] estende a ferramenta EvoSuite para acessar diretamente APIs privadas e criar objetos *mock*. Os resultados experimentais mostram que a cobertura de código e a taxa de detecção de faltas melhoraram. Analisamos que a cobertura do código está diretamente relacionada com a taxa de detecção, pois os sistemas com maiores taxas de coberturas, obtiveram as maiores taxas de detecção de faltas de refatoramento.

Neste trabalho de dissertação, abordamos o problema da validação de edições de refatoramento, a partir de suites de teste JUnit geradas automaticamente. Nos concentramos no tipo de refatoramento *Extract Method*. Considerando as versões mais recentes das ferramentas EvoSuite e Randoop com características aprimoradas que prometem enfrentar os problemas relatados no estudo original. Comparamos os resultados de ambos os estudos e analisamos os fatores que influenciam os resultados. Ao considerar os fatores investigados, desenvolvemos modelos de regressão. Os modelos são avaliados com uma replicação de estudo adicional que considera três novos *softwares open source*.

Capítulo 7

Conclusões

Neste trabalho realizamos dois estudos exploratórios que investigaram a efetividade e as limitações de suites geradas pelas ferramentas de geração automática de testes Randoop e Evosuite, quando usadas para validar faltas de refatoramento do tipo *extract method*.

Os resultados desses estudos mostraram que as suites geradas são limitadas, quanto à métodos não públicos, bem como a complexidade dos objetos recebidos/retornados. Isto na prática, pode levar o desenvolvedor a erroneamente aferir que um refatoramento foi seguro quando na verdade não foi, levando por fim a inserir um *bug* no sistema. As suites Randoop e Evosuite deixaram de detectar, respectivamente, 49,2% e 35,8% das faltas injetadas no primeiro experimento. Quanto analisamos o segundo experimento, já com as versões mais novas das ferramentas, temos que 70% e 30% das faltas não foram detectadas. Número estes que confirma a permanência das referidas limitações nas versões mais atuais do Randoop e do Evosuite.

Diante disso, propusemos três abordagens para a predição da probabilidade de ferramentas de geração automática de testes, gerarem suites efetivas para detecção de falta de refatoramento. Sendo estas abordagens, uma para a ferramenta Randoop, outra para Evosuite e a terceira para Ambas ferramentas, a partir de uma análise combinada. Nossas abordagens baseam-se em três fatores: visibilidade do método, o tipo do parâmetro e o tipo do retorno do método. Porém, com um coeficiênte adicional, pois para cada fator, tem-se um valor ou peso estatítico atribuído ao mesmo.

Para validar nossas abordagens, nós realizamos um terceiro experimento. O qual foi executado sob as mesmas condições de ambiente de desenvolvimento e configurações, dos

demais estudos. Para isto, utilizamos novas unidades experimentais Java, *open source*, que são conhecidas e já utilizadas em outras pesquisas. As três abordagens (Randoop, Evosuite e Ambas ferramentas) obtiveram um desempenho satisfatório, conseguindo prêver corretamente 51,7% , 84,2% e 85,8%, respectivamente.

Além disso, propusemos a REFANALYZER. Que é uma ferramenta cujo intuito é ajudar os desenvolvedores a decidir quando confiar em suites geradas automaticamente, para validar refatoramento do tipo *extract method*. Seu uso pode ser feito de duas formas, na primeira forma REFANALYZER recebe um projeto JAVA e fornece como saída, uma lista de métodos classificados como problemáticos. Na segunda forma, o desenvolvedor fornece a informação de onde o refatoramento foi realizado, a partir dessa informação, a REFANALYZER informa a probabilidade das ferramentas de geração automática de testes, gerarem casos que detecte uma falta, no referido local. Por fim, surgiu a necessidade de validar REFANALYZER, de modo à saber o quão útil nossas proposições podem ser na validação de refatoramento dentro do contexto de projetos ágeis. Com isso, realizamos um estudo com 30 desenvolvedores profissionais, distribuídos em 8 projetos de *software* distintos de grande e médio porte. Este estudo teve como objetivo avaliar a capacidade da ferramenta em ajudar os desenvolvedores a decidir quando usar suites geradas. Para avaliar a perspectiva dos desenvolvedores quanto á utilidade da REFANALYZER, pedimos que avaliassem entre uma das cinco opções: Inútil, Pouco Útil, Neutro, Útil ou Muito Útil. Como resultado, obtivemos 90% de respostas positivas com relação a utilização de REFANALYZER, distribuídas da seguinte forma: 14 participantes avaliaram REFANALYZER como Muito Útil, 13 como Útil, 2 como Neutro, 1 como Pouco Útil e 0 como Inútil.

7.1 Trabalhos Futuros

Como trabalho futuro, planejamos: i) refinar os modelos de predição propostos, especialmente o Randoop, para que eles considerem outros fatores de influência para melhorar suas taxas de predição; ii) Utilizar outras estratégias, como técnicas de aprendizado de máquina, para obter diferentes modelos de predição e/ou refinar os propostos neste artigo; iii) Expandir nossos estudos e modelos, considerando um conjunto maior de edições e faltas de refatoramento; e iv) Propor um conjunto de diretrizes sobre como os desenvolvedores podem lidar

com as limitações das ferramentas e combinar conjuntos de testes manuais e gerados para compor conjuntos de teste mais eficazes.

Bibliografia

[1] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 263–272. IEEE Press, 2017.

[2] Everton LG Alves, Tiago Massoni, and Patrícia Duarte de Lima Machado. Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. *Journal of Systems and Software*, 2016.

[3] Everton LG Alves, Tiago Massoni, and Patrícia Duarte de Lima Machado. Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. *Journal of Systems and Software*, 123:223–238, 2017.

[4] Everton LG Alves, Tiago Massoni, and Patrícia DL Machado. Test coverage and impact analysis for detecting refactoring faults: a study on the extract method refactoring. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1534–1540. ACM, 2015.

[5] Everton LG Alves, Myoungkyu Song, Tiago Massoni, Patricia DL Machado, and Miryung Kim. Refactoring inspection support for manual refactoring edits. *IEEE Transactions on Software Engineering*, 2017.

[6] ZF Andy Field, Jeremy Miles, and Zoe Field. *Discovering statistics using R*. Sage, Thousand Oaks, 2012.

[7] Andrea Arcuri, José Campos, Gordon Fraser, E Daka, J Dorn, W Weimer, and R Abreu. Unit test generation during software development: Evosuite plugins for maven, intellij

and jenkins. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[8] Andrea Arcuri, Gordon Fraser, and René Just. Private api access and functional mocking in automated unit test generation. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 126–137. IEEE, 2017.

[9] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. When does a refactoring induce bugs? an empirical study. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, pages 104–113. IEEE, 2012.

[10] Robert V Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[11] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):5, 2015.

[12] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194. ACM, 2007.

[13] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 389–398. IEEE, 2005.

[14] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[15] Martin Fowler, K Beck, J Brant, W Opdyke, and D Roberts. Refactoring: Improving the design of existing programs, 1999.

[16] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[17] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[18] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[19] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[20] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[21] David G Kleinbaum and Mitchel Klein. Introduction to logistic regression. In *Logistic regression*, pages 1–39. Springer, 2010.

[22] Jeshua S Kracht, Jacob Z Petrovic, and Kristen R Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 256–265. IEEE, 2014.

[23] José Carlos Maldonado, Ana Regina Cavalcanti da Rocha, and Kival Chaves Weber. Qualidade de software: teoria e prática. *São Paulo*, 2001.

[24] Alysson Milanez, Dênnis Sousa, Tiago Massoni, and Rohit Gheyi. Jmlok2: A tool for detecting and categorizing nonconformances. *CBSoft (Tools session)*, pages 69–76, 2014.

[25] Alysson F Milanez, Tiago L Massoni, Rohit Gheyi, and Campina Grande-PB-Brazil. Categorizing nonconformances between programs and their specifications, 2013.

[26] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 93:39–64, 2014.

[27] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[28] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[29] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[30] Erik Poll, Pieter Hendrik Hartel, and Eduard Karel Jong. A java reference model of transacted memory for smart cards. Technical report, Centre for Telematics and Information Technology, University of Twente, 2002.

[31] Napol Rachatasumrit and Miryung Kim. An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 357–366. IEEE, 2012.

[32] Ricardo Miguel Soares Rodrigues. *JML-Based formal development of a Java card application for managing medical appointments*. PhD thesis, Universidade da Madeira, 2009.

[33] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, pages 1–42, 2016.

[34] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.

[35] Indy PSC Silva, Everton LG Alves, and Wilkerson L Andrade. Analyzing automatic test generation tools for refactoring validation. In *Proceedings of the 12th International Workshop on Automation of Software Testing*, pages 38–44. IEEE Press, 2017.

[36] Gustavo Soares. Making program refactoring safer. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 521–522. ACM, 2010.

[37] Gustavo Soares, Diego Cavalcanti, Rohit Gheyi, Tiago Massoni, Dalton Serey, and Márcio Cornélio. Saferefactor-tool for checking refactoring safety. *Tools Session at SBES*, pages 49–54, 2009.

[38] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[39] Chang-ai Sun, Cuiyang Fan, Zhen Wang, and Huai Liu. dμreg: a path-aware mutation analysis guided approach to regression testing. In *Proceedings of the 12th International Workshop on Automation of Software Testing*, pages 59–64. IEEE Press, 2017.

[40] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Precise identification of problems for structural test generation. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 611–620. IEEE, 2011.

[41] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Apêndice A

Análise do Perfil dos Profissionais

Análise do Perfil

1. Qual função está desempenhando atualmente?

_____

2. Qual das seguintes opções descreve sua experiência com programação?

( ) 0-1 ano ( ) 1-3 anos ( ) 3-5 anos ( ) 5-7 anos ( ) Mais de 7 anos

3. Qual das seguintes opções descreve sua experiência no uso do Eclipse IDE?

( ) 0-1 ano ( ) 1-3 anos ( ) 3-5 anos ( ) 5-7 anos ( ) Mais de 7 anos

4. Ao trabalhar em um projeto, com que frequência você realiza e revisa as edições de refatoramento?

( ) Nunca ( ) Diariamente ( ) Uma vez na semana ( ) Uma vez no mês ( ) Raramente

5. Quais os tipos de refatoramento você conhece? (Respostas múltiplas)

( ) Add Parameter ( ) Extract method ( ) Move Method ( ) Push down Method

( ) Decompose Conditional ( ) Inline Class ( ) Pull up Field ( ) Rename Method

( ) Encapsulate Field ( ) Inline Method ( ) Pull up Method ( )Replace Parameter with a Method

( ) Extract Class ( ) Move Field ( ) Push down Field ( ) Substitute Algorithm

6. Como você costuma realizar edições de refatoramento?

( ) Manualmente ( ) Ferramenta. Quais?_____

7. Como você geralmente revisa as edições de refatoramento?

( ) Manualmente ( ) Ferramenta. Quais?_____

8. No projeto onde trabalha, quais tipos de testes são utilizados?

( ) Manuais ( ) Automáticos. Gerados por quais ferramentas?
_____

9. Caso utilize testes automáticos. Você confia plenamente nos testes gerados, caso seja inserido alguma falta durante a realização de um refatoramento?

( ) Sim, confio. ( ) Não confio

Muito Obrigada!

Apêndice B

Avaliando a Ferramenta RefAnalyzer

CENÁRIO 1

Suponha que, a fim de reduzir a duplicidade de código, você tem a intenção de realizar um refatoramento do tipo *extract method* no método a seguir:

```
277   private byte[] process(APDU apdu)
278   {
279 // After the applet is successfully selected, the JCRE dispatches incoming
280 // APDUs to the process method.   At this point, only the first 5 bytes
281 // [CLA, INS, P1, P2, LC] are available in the APDU buffer.
282       byte[] buffer = apdu.getBuffer();
283 // The JCRE also passes the SELECT APDU commands to the applet
284 // (which is ignored)
285       if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
286           (buffer[ISO7816.OFFSET_INS] == (short)(0xA4)) )
287       ISOException.throwIt(SW_IGNORED);
288 // Check CLA
289       if (buffer[ISO7816.OFFSET_CLA] != Mondex_CLA)
290      ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
291 // Ignores message not sent to this purse
292       if (Util.getShort(buffer, ISO7816.OFFSET_P1) != name)
293      ISOException.throwIt(SW_IGNORED);
294 // Calls the method indicated by the INS byte
295       switch (buffer[ISO7816.OFFSET_INS])
296       {
297       case StartFrom: start_from_operation(apdu); break;
298       case StartTo: start_to_operation(apdu); break;
299       case Req: req_operation(apdu); break;
300       case Val: val_operation(apdu); break;
301       case Ack: ack_operation(apdu); break;
302       case ReadExLog: read_ex_log_operation(apdu); break;
303       case ClearExLog: clear_ex_log_operation(apdu); break;
304       default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
305       }
306
307       return buffer;
308
309   }
310
```

Como o sistema em questão é bastante estável, é necessário garantir que seu refatoramento não impacte nas funcionalidades do mesmo.

INFORMAÇÃO OFERECIDA:

Um modelo preditor afirma que, para o método em questão, em ***48%*** dos casos, as ferramentas de geração de testes irão criar suítes efetivas para detectar faltas de refatoramento.

Questões Cenário 1

De posse dessas informações, responda as questões abaixo:

1. Como você avalia a informação fornecida?

( ) Muito Útil ( ) Útil ( ) Neutro ( ) Pouco Útil ( ) Inútil

Comente:

_____
_____
_____

2. Sobre a aplicação do refatoramento você...

( ) Seguiria com o refatoramento.
( ) Desistiria do refatoramento.
( ) Postergaria o refatoramento para um outro momento.

Comente:

_____
_____
_____

3. Com relação a efetividade do uso de suítes geradas para esse contexto, você...
( ) Confia plenamente.
( ) Confia parcialmente.
( ) Tem opinião neutra.
( ) Não confia.

Comente:

_____
_____
_____

4. Sobre a utilização de testes para a validação do refatoramento, você...
( ) Faria uso de suítes geradas.
( ) Faria uso de suítes geradas, mas combinadas com alguma outra estratégia automática (e.g., análise estática de código)
( ) Faria uso de suítes geradas, mas combinadas com testes criados manualmente
( ) Faria uso somente de testes criados manualmente

Comente:

_____
_____
_____

CENÁRIO 2

Suponha que, a fim de reduzir a duplicidade de código, você tem a intenção de realizar um refatoramento do tipo *extract method* no método a seguir:

```
53  public byte addMedicine (byte appointmentID, byte diagnosticID, byte treatmentID) {
54      boolean notInUse = true;
55      byte medicineID = 0;
56      for(byte code = 0x00; code <= 0x7F; code++){
57          for (short j = (short)0; j < medicines.length; j++) {
58              if(medicines[j] != null){
59                  if(medicines[j].getMedicineID() == code
60                          && medicines[j].getAppointmentID()== appointmentID
61                          && medicines[j].getDiagnosticID()== diagnosticID
62                          && medicines[j].getTreatmentID() == treatmentID){
63                      notInUse = false;
64                      break;
65                  }
66              }
67          }
68          if(notInUse){
69              medicineID = code;
70              break;
71          }
72          notInUse = true;
73      }
74
75      for (short i = (short)0; i < (short)MedicinesSetup.MAX_MEDICINE_ITEMS; i++) {
76          if (medicines[i] == null) {
77              Medicine m = new Medicine_Impl(appointmentID, diagnosticID, treatmentID, medicineID);
78              medicines[i] = m;
79              break;
80          }
81      }
82      return medicineID;
83  }
```

Como o sistema em questão é bastante estável, é necessário garantir que seu refatoramento não impacte nas funcionalidades do mesmo.

INFORMAÇÃO OFERECIDA:

Um modelo preditor afirma que, para o método em questão, em ***72%*** dos casos, as ferramentas de geração de testes irão criar suítes efetivas para detectar faltas de refatoramento.

Questões Cenário 2

De posse dessas informações, responda as questões abaixo:

1. Como você avalia a informação fornecida?

( ) Muito Útil ( ) Útil ( ) Neutro ( ) Pouco Útil ( ) Inútil

Comente:

_____
_____
_____

2. Sobre a aplicação do refatoramento você...

( ) Seguiria com o refatoramento.
( ) Desistiria do refatoramento.
( ) Postergaria o refatoramento para um outro momento.

Comente:

_____
_____
_____

3. Com relação a efetividade do uso de suítes geradas para esse contexto, você...
( ) Confia plenamente.
( ) Confia parcialmente.
( ) Tem opinião neutra.
( ) Não confia.

Comente:

_____
_____
_____

4. Sobre a utilização de testes para a validação do refatoramento, você...
( ) Faria uso de suítes geradas.
( ) Faria uso de suítes geradas, mas combinadas com alguma outra estratégia automática (e.g., análise estática de código)
( ) Faria uso de suítes geradas, mas combinadas com testes criados manualmente
( ) Faria uso somente de testes criados manualmente

Comente:

_____
_____
_____

CENÁRIO 3

Suponha, ainda para o JMock, que uma ferramenta lhe forneça a lista de métodos cujas ferramentas de geração de testes tendem a não ser efetivas para detecção de faltas de refatoramento.

```
------------------------------------------------------------
====== Métodos problemáticos para as ferramentas=========
============================================================

- Nome do Metodo:addMatcher,    - Assinatura: private org.jmock.builder.InvocationMockerBuilder org.jmock.builder.InvocationMockerBuilder.addMatcher(org.jmock.core.InvocationMatcher)
- Nome do Metodo:newCoreMock,    - Assinatura: protected org.jmock.core.DynamicMock org.jmock.cglib.MockObjectTestCase.newCoreMock(java.lang.Class,java.lang.String,java.lang.Class[],java.lang.Object[])
- Nome do Metodo:newCoreMock,    - Assinatura: protected org.jmock.core.DynamicMock org.jmock.cglib.MockObjectTestCase.newCoreMock(java.lang.Class,java.lang.String)
- Nome do Metodo:hiddenInvocationMocker,    - Assinatura: private org.jmock.core.InvocationMocker org.jmock.core.AbstractDynamicMock.hiddenInvocationMocker(java.lang.String,org.jmock.core.InvocationMatcher,org.jmock.core.Stub)
- Nome do Metodo:dispatchOrder,    - Assinatura: protected abstract java.util.Iterator org.jmock.core.AbstractInvocationDispatcher.dispatchOrder(java.util.List)
- Nome do Metodo:getReadMethod,    - Assinatura: private java.lang.reflect.Method org.jmock.core.constraint.HasPropertyWithValue.getReadMethod(java.lang.Object) throws java.beans.IntrospectionException
- Nome do Metodo:relationship,    - Assinatura: protected java.lang.String org.jmock.core.constraint.StringContains.relationship()
- Nome do Metodo:relationship,    - Assinatura: protected java.lang.String org.jmock.core.constraint.StringEndsWith.relationship()
- Nome do Metodo:relationship,    - Assinatura: protected java.lang.String org.jmock.core.constraint.StringStartsWith.relationship()
- Nome do Metodo:relationship,    - Assinatura: protected abstract java.lang.String org.jmock.core.constraint.SubstringConstraint.relationship()
- Nome do Metodo:dispatchOrder,    - Assinatura: protected java.util.Iterator org.jmock.core.FIFOInvocationDispatcher.dispatchOrder(java.util.List)
- Nome do Metodo:toJavaSyntax,    - Assinatura: private static java.lang.String org.jmock.core.Formatting.toJavaSyntax(char)
- Nome do Metodo:toJavaSyntax,    - Assinatura: private static java.lang.String org.jmock.core.Formatting.toJavaSyntax(java.lang.String)
- Nome do Metodo:makeBoxTypesMap,    - Assinatura: private static java.util.Map org.jmock.core.Invocation.makeBoxTypesMap()
- Nome do Metodo:dispatchOrder,    - Assinatura: protected java.util.Iterator org.jmock.core.LIFOInvocationDispatcher.dispatchOrder(java.util.List)
- Nome do Metodo:loadObject,    - Assinatura: private org.jmock.examples.timedcache.TimedCache$TimestampedValue org.jmock.examples.timedcache.TimedCache.loadObject(java.lang.Object)
- Nome do Metodo:getActualCollection,    - Assinatura: protected abstract java.util.Collection org.jmock.expectation.AbstractExpectationCollection.getActualCollection()
- Nome do Metodo:getExpectedCollection,    - Assinatura: protected abstract java.util.Collection org.jmock.expectation.AbstractExpectationCollection.getExpectedCollection()
- Nome do Metodo:getActualCollection,    - Assinatura: protected java.util.Collection org.jmock.expectation.ExpectationList.getActualCollection()
- Nome do Metodo:getExpectedCollection,    - Assinatura: protected java.util.Collection org.jmock.expectation.ExpectationList.getExpectedCollection()
- Nome do Metodo:getActualCollection,    - Assinatura: protected java.util.Collection org.jmock.expectation.ExpectationSet.getActualCollection()
- Nome do Metodo:getExpectedCollection,    - Assinatura: protected java.util.Collection org.jmock.expectation.ExpectationSet.getExpectedCollection()
- Nome do Metodo:addNewInvocationMocker,    - Assinatura: private org.jmock.builder.NameMatchBuilder org.jmock.Mock.addNewInvocationMocker()
- Nome do Metodo:newCoreMock,    - Assinatura: protected org.jmock.core.DynamicMock org.jmock.MockObjectTestCase.newCoreMock(java.lang.Class,java.lang.String)
- Nome do Metodo:times,    - Assinatura: private org.jmock.core.InvocationMatcher test.jmock.examples.timedcache.TimedCacheTest.times(int)
- Nome do Metodo:returnValues,    - Assinatura: private org.jmock.core.Stub test.jmock.examples.timedcache.TimedCacheTest.returnValues(java.lang.Object,java.lang.Object)
- Nome do Metodo:returnValues,    - Assinatura: private org.jmock.core.Stub test.jmock.examples.timedcache.TimedCacheTest.returnValues(java.lang.Object,java.lang.Object,java.lang.Object)

--> NOTA IMPORTANTE: Os métodos abaixo possuem características de que dificultam a geração automática de testes para estes,
onde estas podem está relacionadas com a visibilidade do mesmo, os parametros de entrada e/ou tipo de retorno.
```

De posse dessa informação, responda as questões abaixo:

1. Como você avalia a informação fornecida?

( ) Muito Útil ( ) Útil ( ) Neutro ( ) Pouco Útil ( ) Inútil

Comente:

_____
_____
_____

2. Quais ações/atitudes você tomaria a partir da informação fornecida?

_____
_____
_____
_____